uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería de Sistemas Audiovisuales
2017-2018

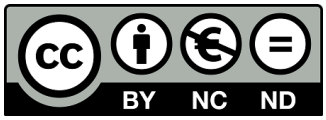
Trabajo Fin de Grado

"Diseño de un Sistema de Recomendación basado en AI para LoL"

Sergio Elola García

Tutor:
Jesús Cid-Sueiro

Leganés, 5 de octubre de 2018



ABSTRACT

League of Legends (LoL) is one of the most played video games in the world, having a monthly amount of more than 100 million players. In every competitive LoL match, there is a pre-game phase in which players must make important decisions that can affect the outcome of the game.

In this project, a recommendation system is proposed to help the players in that decision making. This system is designed so that the recommendation can be provided in real time. In order to achieve it, a predictor of the winner team of a game is built following different Machine Learning strategies. This capacity of prediction is exploited, using Game Theory techniques, to get a recommendation for any player of the game.

The work done for this project and the resulting algorithm can inspire the development of other recommendation systems in other games with similar pre-games phases. A English summary of the project is provided as an annex.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1. Marco y motivación del proyecto

La importancia de la industria de los videojuegos es cada día mayor. Según datos de la Asociación Española de Videojuegos (AEVI), la facturación mundial del mercado del videojuego en 2016 fue aproximadamente de 100.000 millones de dólares. Según la misma asociación, solo en España la industria de los videojuegos facturó en ese mismo año 1163 millones de euros, lo que representa el 0.11 % del PIB español. Tal es la relevancia de este mercado, que lleva varios años liderando el sector de ocio audiovisual e interactivo en España, por encima de sectores como el cine y la música.

Si bien estos datos son sorprendentes, el hecho más significativo a destacar es el gran crecimiento de esta industria. Según estimaciones de Newzoo [1], para 2021 la recaudación mundial estará alrededor de los 180.000 millones de dólares o, lo que es lo mismo, esta industria experimentará un crecimiento del 11 %. La mayor parte de este incremento viene dada por los juegos móviles (tanto smartphones como tablets), que en 2021 supondrán el 59 % de este mercado. Sin embargo, existen otros factores por los que esta industria gana día a día más usuarios, de los que destacan los Deportes Electrónicos, en adelante eSports, en torno a los cuales gira este proyecto.

Los eSports se refieren comúnmente a videojuegos competitivos, en distintos dispositivos electrónicos, que están coordinados por diferentes ligas y torneos, y donde los jugadores profesionales pertenecen a equipos u otras organizaciones deportivas que están patrocinadas [2]. La industria de los eSports es reciente: se estima que la economía global de los eSports alcanzará los 905 millones de dólares en 2018, lo que supone un crecimiento con respecto al año anterior del 38 % [3]. Sin embargo, los datos más interesantes sobre los eSports son aquellos relacionados con los espectadores. Se estima que la audiencia total en eSports en 2017 fue de 335 millones de espectadores. Como dato sugerente, la final del campeonato de mitad de temporada ("Mid-Season Invitational") de League of Legends de 2018 tuvo 60 millones de espectadores únicos, es decir, casi 3 veces más que la final de la NBA.

Estas cifras tan altas se deben, en gran medida, a la existencia de plataformas de streaming de vídeo en vivo como Twitch, la más popular. Estas plataformas, junto con las redes sociales, han llevado a que los eSports sean un mercado con grandes expectativas que atrae a profesionales de distintas industrias, desde clubes de fútbol (Paris Saint Germain, Schalke 04, Valencia CF...) hasta operadoras de telecomunicaciones (Orange, Vodafone, Movistar...).

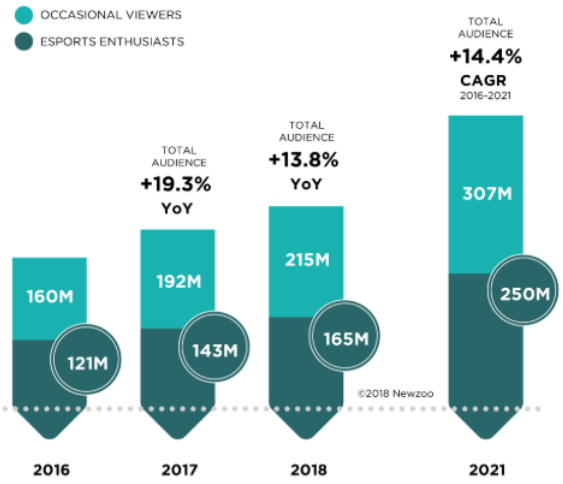


Figura 1.1: Crecimiento de la audiencia de eSports. Obtenida de [1]

Dentro de los eSports, podemos encontrar el juego de ordenador League of Legends, habitualmente llamado por sus siglas, "LoL". Aunque la empresa desarrolladora de League of Legends, Riot Games, no saca datos oficiales desde 2014, donde el juego ya tenía 27 millones de usuarios mensuales, se cree con cierta confianza que en 2016 se sobrepasó la cifra de 100 millones de jugadores mensuales [4]. LoL lleva siendo, en el momento de escribir este documento, el juego más jugado del mundo desde 2012 y desde entonces domina claramente la industria de los eSports.

Una de las razones de su éxito es, a diferencia de otros eSports, el gran apoyo que da Riot Games a las competiciones de LoL, siendo la propia empresa la que organiza la mayor parte de eventos. Por ejemplo, en el campeonato mundial del año 2017, el evento más importante del año, se repartieron más de 4 millones de euros en premios. Aunque en la mayor parte de los casos los premios van dirigidos a los jugadores, estos no son los únicos que forman parte de un equipo. Los equipos profesionales tienen detrás una serie de profesionales que velan por el éxito del equipo: entrenador, analistas, mánager, psicólogo, etc. Todo esto no hace más que motivar a los jugadores corrientes a jugar al LoL, no solo por diversión, sino porque existe la posibilidad de poder ganarse la vida jugando a videojuegos, algo que era impensable hace una década.

Dentro del propio juego, en cada partida de League of Legends existe una fase llamada Draft en la que los jugadores deben tomar decisiones que pueden ser clave para el desarrollo de la partida. Dichas decisiones son un problema complejo y no hay herramientas automáticas que faciliten esa toma de decisión de manera formal, ya que en la mayor parte de los casos estas decisiones se toman por intuición adquirida a través de la experiencia en el juego.

Este tipo de fases son difíciles de formalizar, es decir, no hay método ni fórmula alguna que permita obtener la mejor decisión en cada caso. Sin embargo, existen

abundantes datos que potencialmente permitirían aprender de manera inductiva a partir de los resultados de partidas anteriores. La motivación de este proyecto es explorar esa posibilidad, con el fin elaborar un sistema que pueda ayudar a los jugadores corrientes de LoL a tomar mejores decisiones en sus partidas.

1.2. Objetivos del proyecto

1.2.1. Sobre League of Legends

Antes de entrar directamente con los objetivos, es necesario primero describir brevemente el juego para que aquellos lectores que no lo conozcan puedan entender lo que se pretende hacer. League of Legends es un juego gratuito de ordenador, que se encuentra dentro del género MOBA (Multiplayer Online Battle Arena). En este tipo de juegos, equipos de jugadores luchan entre sí en un recinto cerrado llamado arena o mapa, y el objetivo es destruir la estructura principal del equipo enemigo (el "Nexo", en el caso de LoL).

Durante la partida se generan periódicamente en cada equipo pequeñas unidades de ayuda controladas por inteligencia artificial que marchan hacia la estructura principal del rival a través de los distintos caminos que hay, que se suelen llamar calles o, en inglés, "lanes". En las calles hay varias torres defensivas que hay que destruir para poder llegar hasta la estructura principal. Además, cada jugador controla un personaje al que puede ir mejorando durante la partida, contribuyendo las mejoras a la estrategia global de esta.

En LoL hay dos equipos (equipo azul y equipo rojo) con 5 jugadores por equipo, en el que cada uno controla un personaje diferente, denominado "campeón". Cada campeón tiene sus propias habilidades y estadísticas, que son valores que indican como de bueno es un campeón haciendo cierta cosa (por ejemplo, cantidad de vida o daño). Hasta la fecha de escritura de este proyecto existen 141 campeones diferentes. Asimismo, en cada equipo, cada jugador tiene un rol diferente, y aunque el juego está en contaste cambio y los roles pueden cambiar, se suele mantener la misma distribución de los jugadores por el mapa.

Para entender en que consisten los roles en el LoL, utilicemos como analogía el fútbol. De la misma manera que en el fútbol existen por ejemplo delanteros o un portero, que tienen su función y posición dentro del campo, en el LoL los jugadores también se distribuyen estratégicamente por el mapa y tienen su objetivo dentro de la partida. Normalmente en cada equipo, el jugador llamado "Top" va a la calle superior (Top Lane), el "Mid" va a la calle central (Middle Lane), los jugadores "Support" y "Marksman" van a la calle inferior (Bottom Lane) y el quinto jugador se mueve por la jungla, que es la zona que hay entre las calles, y se denomina "Jungle".

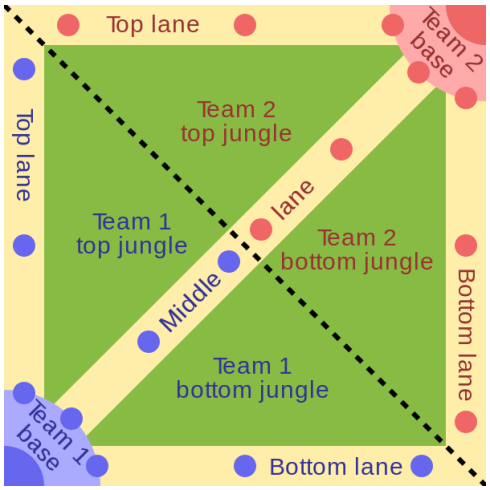


Figura 1.2: Mapa simple de LoL. Representadas en amarillo las distintas calles y en verde la jungla. Los puntos representan las torres defensivas de cada equipo. Obtenida de [5]

Al igual que en el fútbol un defensa puede participar en un contraataque o un delantero puede bajar a defender, los jugadores no permanecen toda la partida en sus respectivas calles, sino que se mueven por todo el mapa. Aunque no es necesario entrar en detalle y explicar la función específica de cada rol dentro de la partida para este proyecto, sí que es importante saber de su existencia ya que se tendrán en cuenta a la hora de desarrollar el sistema de recomendación. La información proporcionada será suficiente para entender perfectamente el presente proyecto pero, si algún lector quisiera profundizar más sobre LoL, se recomienda la lectura de [6].

Antes de hablar del Draft, sobre el cual va a girar todo el proyecto, es importante comentar que existe una competición interna en LoL en el la que los jugadores se clasifican en distintas ligas (bronce, plata, oro...), según lo buenos y habilidosos que sean en el juego. De esta manera, cada jugador competirá con otros jugadores de su misma capacidad, lo que hace que las partidas estén más igualadas y por tanto que el juego sea más entretenido. La aspiración de los jugadores, como en cualquier otro deporte o juego, es llegar a la clasificación más alta posible.

Draft o Selección de Campeones

League of Legends tiene varios modos de juego. Sin embargo, el modo al que nos vamos a referir durante todo el proyecto es el competitivo oficial, en el cual hay un proceso previo a la propia partida que se llama Selección de Campeones o, más conocido, “Draft”. Durante este proceso hay dos fases: en primer lugar la de fase “Bans”, donde cada uno de los 10 jugadores inhabilita a un campeón para que no pueda ser jugado por ningún jugador de ningún equipo (a ese campeón se le llama “ban” y se dice que está baneado), y en segundo lugar la fase de “Picks”, donde cada jugador selecciona el campeón con el que jugará durante toda la partida, que

no podrá ser ninguno de los campeones baneados o ya seleccionados.

En la fase de Picks, los campeones se seleccionan en un determinado orden simétrico, de manera que ambos equipos estén en igualdad de condiciones. Es decir, este orden permite a los equipos hacer selecciones estratégicas de modo que puedan contrarrestar los campeones seleccionados por el otro equipo. El orden de cada jugador es aleatorio para cada partida. En la figura 1.3 viene representado la fase de picks: primero elige el primer jugador del equipo azul, a continuación los dos primeros del equipo rojo, luego los dos siguientes del equipo azul, después los dos siguientes del equipo rojo, posteriormente los dos siguientes del equipo azul y por último el jugador restante del equipo rojo.

El Draft es una fase muy importante en una partida. Prácticamente la totalidad de analistas y jugadores profesionales están de acuerdo en que una partida se puede ganar o perder en la selección de campeones. Si un equipo tiene campeones muy fuertes o contrarresta muy bien al equipo contrario, tendrá muchas más posibilidades de ganar la partida. A pesar de que la elección del campeón puede depender de muchos factores, los jugadores normalmente se basan en su propia experiencia en el juego y sus preferencias para tomar decisiones que son claves para la partida. De esta falta de herramientas robustas surge la necesidad de crear un sistema de recomendación que se base en datos de partidas anteriores para proporcionar ayuda a los jugadores durante esta fase.

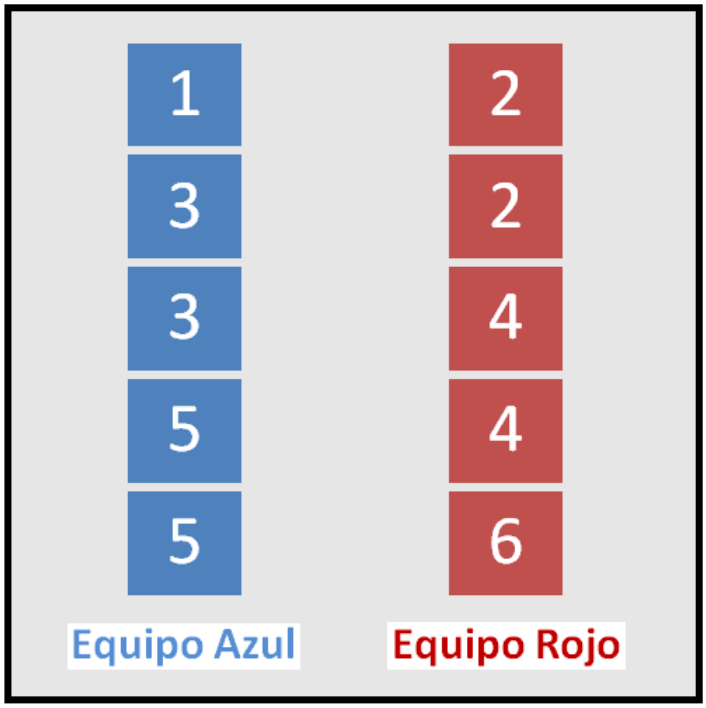


Figura 1.3: Representación gráfica de la fase de Picks en un Draft de League of Legends. Los rectángulos representan a los jugadores y los números representan el orden (turno) en el cual selecciona cada jugador.

1.2.2. Objetivos

El objetivo de este proyecto es diseñar un sistema que recomiende, durante la fase de Draft de una partida de League of Legends, el campeón o campeones con los que el jugador tenga más probabilidad de ganar la partida. Para ello, se tendrán en cuenta los campeones que se hayan seleccionado previamente en ese Draft, tanto campeones aliados como enemigos.

El sistema de recomendación servirá para cualquier posición que el jugador tenga en el orden de selección del Draft. Además, la recomendación se ajustará al nivel que tenga el jugador (es decir, lo bueno que sea jugando, que dependerá de la liga en la que esté clasificado) y del rol (posición) que vaya a jugar en esa partida.

Para desarrollar el sistema se usarán datos extraídos específicamente para este proyecto a través de la API de Riot Games. Además, durante todo el proyecto se utilizará Python, uno de los lenguajes de programación más importantes en desarrollo de Inteligencia Artificial y que será de mucha utilidad gracias a la gran cantidad de bibliotecas existentes.

Hoja de ruta del proyecto

Para lograr hacer el sistema de recomendación se necesita tener la capacidad de predecir la probabilidad de ganar de cada jugador en función de los campeones. Por ello, buena parte del proyecto se dedica a hacer la elaboración de un predictor. Este predictor será capaz de predecir el ganador de una partida dados los 10 campeones (5 en cada equipo) que forma una partida y proporcionará la probabilidad de de ganar de cada equipo. Esas probabilidades serán claves para, haciendo uso de ciertas técnicas propias de Teoría de Juegos, lograr hacer la recomendación para un único jugador.

La predicción del equipo ganador es la parte que mayor dificultad entraña y la que requerirá más tiempo, por lo que se decide separar en dos capítulos distintos, siguiendo una estrategia de las siguientes en cada uno:

- Machine Learning Clásico: Se ha llamado así a aquellas aproximaciones del Aprendizaje Automático que no incluyan Aprendizaje Profundo (Deep Learning).
- Deep Learning: A pesar de que el Deep Learning sea en sí una rama de Machine Learning, se ha decidido separar ambas áreas porque se procederá de distinta manera.

La principal razón por la que la predicción entraña tanta dificultad es porque la desarrolladora del juego pretende, lógicamente, hacer un juego equilibrado en el que

las partidas estén lo más igualadas posible. Hay muchos profesionales que trabajan exclusivamente para que este equilibrio se mantenga, por lo que no se espera obtener una precisión alta en el predictor.

1.3. Estructura del proyecto

La memoria de este proyecto se compone de ocho capítulos que se describen a continuación:

1. *Introducción*: Se presenta la motivación del proyecto, se hace una breve explicación del tema a tratar, se definen los objetivos a alcanzar y la estructura del proyecto.

2. *Estado del Arte*: Estudio de las líneas de investigación existentes y trabajos similares.

3. *Variedad, extracción y distribución de los datos*: Se describe el tipo de datos que se tienen, cómo se hace la extracción de los datos con la API y cómo se distribuirán los datos para hacer el entrenamiento y prueba del clasificador.

4. *Predicción con Machine Learning Clásico*: Se procede a la predicción del equipo ganador de una partida con técnicas del llamado en este proyecto como Machine Learning Clásico.

5. *Predicción con Deep Learning*: Predicción del equipo ganador con Deep Learning.

6. *Algoritmo de recomendación*: Elaboración del algoritmo para el sistema de recomendación.

7. *Conclusiones, Líneas Futuras*: Donde se indican las aportaciones del proyecto y las posibles líneas de investigación a seguir en el futuro.

8. *Marco Regulador y Entorno Socio-Económico*: Se analizan las regulaciones que aplican al problema, el impacto socio-económico (que ya se trató en detalle en el marco y motivación del proyecto) y el presupuesto del proyecto.

2. ESTADO DEL ARTE

2.1. Líneas de investigación en videojuegos RTS

Como se comentó, League of Legends pertenece al género de videojuegos MOBA, que a su vez es un subgénero de los videojuegos RTS (Real Time Strategy). Los juegos RTS, como su propio nombre sugiere, se caracterizan por ser juegos de estrategia que se desarrollan en el tiempo de forma continua para los jugadores, evitando los turnos. Esto hace que este tipo de juegos ofrezca una gran variedad de retos fundamentales en investigación de Inteligencia Artificial (IA), que pueden tener aplicaciones fuera del dominio de los videojuegos [7]. Aunque tradicionalmente la IA en videojuegos estaba relacionada con personajes no jugadores (NPCs en inglés) o con búsquedas de rutas, existen numerosas aplicaciones de IA para juegos RTS, por lo que en realidad este es un campo reciente de investigación. A continuación, basándonos en los artículos de Michael Buro [8] y Raúl Lara-Cabrera et al. [7], se presentan los retos de investigación en IA relacionados los juegos RTS.

Gestión de recursos

La mayoría de juegos RTS incluyen recursos, que sirven a los jugadores para crear o mejorar unidades, armas, objetos o desarrollar tecnologías. Por ejemplo, en el LoL existe el oro, con el que los jugadores pueden comprar distintos objetos para mejorar sus estadísticas o ganar habilidades adicionales que les permite aumentar su poder durante la partida. Una buena gestión de recursos es una parte fundamental de cualquier estrategia. En [9], los autores desarrollaron una herramienta de planificación online para la producción de recursos en el videojuego *Wargus*.

Modelado de oponentes

Los jugadores tienen la habilidad de analizar las acciones del enemigo y aprender de sus fortalezas y debilidades para explotarlas en futuras partidas. Es necesario predecir comportamientos en base a observaciones previas y conseguir así ventaja frente al contrincante. Este reto no es exclusivo de juegos RTS y se pueden aplicar técnicas similares a otros tipos de juegos, dado que modelar el oponente siempre es algo útil. Ejemplos de modelado de oponentes pueden encontrarse en [10] [11] [12] [13] .

Toma de decisiones en condiciones de incertidumbre

Durante muchos momentos de la partida los jugadores desconocen la posición de los enemigos y sus intenciones. Si no disponen de esa información, los jugadores deben hacer suposiciones y actuar en consecuencia. Este reto, que es particularmente característico de este tipo de videojuegos, supone emplear técnicas de predicción y estimación.

Razonamiento espacio-temporal

Los mapas de los juegos RTS y las relaciones temporales de los distintos eventos que se producen en la partida son elementos muy importantes para el desarrollo del juego. En LoL, un buen ejemplo serían los monstruos que salen a lo largo del mapa y que, después de matarlos, te dan oro y/o mejoras de estadísticas, por lo que controlar la zona y el tiempo de su aparición es algo indispensable. En general, la habilidad de aprender y optimizar los conocimientos espacio-temporales en juegos RTS se convierte en algo vital. En [14], los autores estudian como el razonamiento espacial puede mejorar la IA en juegos de estrategia.

Colaboración

En varios juegos RTS, como es el caso del LoL, se forman equipos para unir fuerzas e inteligencia. Como coordinar acciones de manera efectiva entre los aliados tiene que tenerse en cuenta para tener éxito en la partida, lo que supone un reto importante.

Planificación adversaria en tiempo real

La acciones suceden en tiempo real y el entorno es dinámico, hostil e inteligente. Los jugadores tienen que tomar decisiones en un intervalo muy corto de tiempo. En otro tipo de juegos, como puede ser el ajedrez, también se tiene que lidiar con entornos dinámicos e inteligentes, pero el hecho de no estar tan limitados en el tiempo lo hace menos complicado que en el caso de juegos RTS. [15] [16] [17] [18]

2.2. Categorización del proyecto

El presente trabajo no se centra en aplicar técnicas de Inteligencia Artificial en la propia partida, sino en la fase pre-partida donde cada jugador elige su propio campeón teniendo en cuenta sus aliados y enemigos. Aunque no podamos considerar esta fase técnicamente como en tiempo real, ya que se hace por turnos (con un cierto límite de tiempo por turno), sí podríamos catalogar este trabajo como un reto

de *Modelado de Oponentes*. La razón es simple: las decisiones tomadas durante la selección de campeón son muy importantes de cara a la partida.

Por ello, es útil seleccionar el mejor campeón para cada situación teniendo en cuenta el modelado de oponentes hecho a partir de partidas previas (es decir, con la predicción del equipo ganador). Además, como también se tienen en cuenta las elecciones de los campeones aliados, este reto podría considerarse en cierto modo como un problema de *Colaboración* porque, al fin y al cabo, el objetivo es formar el mejor equipo posible para vencer al enemigo.

2.3. Trabajos relacionados con la predicción del equipo ganador

La predicción del equipo ganador, dadas unas características (features) de cada equipo, es claramente el paso más importante de cara a elaborar el sistema de recomendación. Podemos destacar el artículo de Hao Yi Ong et al. [19], en el que los autores intentan obtener la composición de equipo de LoL óptima prediciendo el equipo ganador. Para ello usan como características los campeones seleccionados y varías estadísticas de los jugadores como indicadores de rendimiento o número de partidas ganadas. Para la selección de estadísticas (feature selection) utilizan K-means y DP-means, y para la predicción prueban varios modelos de clasificación: Regresión Logística (LR), Análisis Discriminante Gausiano (GDA) y Máquinas de vectores de soporte (SVM). Consiguen una predicción de alrededor del 70 %, usando la selección de características con K-means y la predicción con SVM o GDA (requiriendo mucho más tiempo SVM). De forma similar, en [20], Jihan Yi usa bastantes más características distintas (ratio de victorias de cada campeón, veces que ha ganado cada jugador en las anteriores 2 o 15 partidas, liga a la que pertenece cada jugador...) para predecir el equipo ganador con algoritmos de clasificación de Machine Learning. Consigue una predicción aproximada del 60 % con Random Forests y Gradient Boosting.

Los artículos mencionados tratan de predecir el resultado de la partida teniendo en cuenta cualquier tipo de dato que puedan obtener de cualquier fuente (desde datos obtenidos de la API hasta cualquier página que ofrezca estadísticas como, por ejemplo, la gran conocida "op.gg"). No obstante, como se pretende hacer un sistema de recomendación en tiempo real, en este trabajo solo se tendrán en cuenta los datos que se pueden obtener en tiempo real durante el Draft. En esta fase, como se explicó, solo se ven los campeones baneados, la elección de campeones aliados y enemigos, y el nombre de los jugadores aliados. Como no se permite ver el nombre de los jugadores enemigos, no se pueden usar ningún dato de ellos como características para la predicción del equipo ganador.

Al no poder usar datos personales de los jugadores enemigos, se decide no usar

los datos de los jugadores aliados. La razón es simple: al entrenar nuestro modelo de predicción no podemos añadir datos adicionales de jugadores a un solo equipo, ya que la intención es que el modelo sea simétrico. Por ejemplo, imaginemos que predecimos que en una cierta partida gana el equipo 1. Cuando decimos que el modelo tiene que ser simétrico, significaría que en este caso si los miembros del equipo 1 hubiesen estado en el equipo 2, y los del equipo 2 en el equipo 1, el modelo tendría que haber predicho que gana el equipo 2. Como consecuencia, por decisión de diseño no se añadirá información adicional para un solo equipo. Por su parte, los campeones baneados solo servirán para no tenerlos en cuenta en la recomendación.

Por tanto, nuestra predicción del equipo ganador se basará exclusivamente en los campeones seleccionados de cada equipo (cada uno con sus estadísticas fijas: vida, poder de ataque, resistencias...). Otra opción sería también incluir estadísticas que ofrecen ciertos sitios webs (como por ejemplo el porcentaje de victoria que tiene cada campeón, qué campeones son los más jugados...). Sin embargo, como este tipo de datos habría que obtenerlos en tiempo real y de sitios externos que no aseguran rigurosidad, se prefiere no usar este tipo de datos y se deja como una posible mejora del sistema de recomendación.

Otros trabajos realizados en LoL tienen que ver con la predicción del equipo ganador durante la propia partida, es decir, no en la selección de campeones sino a medida que avanza la partida. Evidentemente, cuanto más tiempo transcurra en la partida, más fácil será predecir el equipo ganador ya que se dispone de más información. La principal utilidad o aplicación de este tipo de predicción podría ser facilitar al jugador un valor numérico que le convenciese de rendirse para no perder el tiempo (ya que a partir del minuto 15, los equipos pueden elegir rendirse y terminar la partida al instante con una derrota) o, desde otro punto de vista, hacerle ver que la partida no está tan perdida como pudiese pensar. Por ejemplo, en [21] se predice el equipo ganador en el minuto 10 de la partida mediante aprendizaje supervisado, usando la gran cantidad de información disponible en ese minuto, y se determina qué factores son los más relevantes para ganar.

De manera similar, Ben Fradet [22] publicó en su blog el cliente que creó durante el Hackaton de 2016 de Riot Games (la empresa desarrolladora de LoL) para predecir en tiempo real la probabilidad de ganar de cada equipo, usando el framework Spark. Este tipo de predicciones en tiempo real son también comunes en otros juegos RTS. Por ejemplo, para el popular juego *Starcraft*, ejemplos de predicción del ganador pueden encontrarse en [23] [24] donde se usan modelos similares a los usados en los trabajos de LoL, como árboles de decisión, modelos probabilísticos, SVM, etc.

2.4. Trabajos relacionados con sistemas de recomendación en juegos

En el momento de realizar esta memoria, se ha localizado el artículo [25] (ya que su publicación es reciente), en el que los autores proponen (sin implementarlo) un procedimiento para la composición de equipos en LoL de manera muy similar a la que se pretende hacer en este proyecto: construir un árbol de juego que modele todas las combinaciones de selección posibles, utilizar Regresión Lineal para desarrollar una función de predicción que evalúe todas las hojas del árbol y usar el algoritmo minimax con poda alfa-beta para optimizar la recomendación y minimizar la pérdida. La diferencia con respecto a este artículo es que en este proyecto el objetivo es estudiar distintos modelos para el predictor y seleccionar el mejor, y diseñar un algoritmo final siguiendo la misma estrategia (árbol de juego y método minimax), pero que sea óptimo y que pueda proporcionar una recomendación en tiempo real durante la fase de Draft. Al haber tantos campeones (alrededor de 140), suponiendo que 10 campeones han sido baneados, el árbol de juego tendría 130! (130 factorial) posibles combinaciones, una cantidad exageradamente grande, por lo que el diseño del algoritmo requiere un estudio más profundo que el que se hace en ese artículo.

En vez de usar el algoritmo minimax, en [26] se modela la fase de Draft del juego Dota 2 (que es juego MOBA con una fase de selección de personajes muy similar a la del LoL) como un juego combinatorio y se usa el Árbol de Búsqueda Monte Carlo para estimar los valores de las distintas combinaciones de campeones. Además, para el predictor se prueban distintos modelos de clasificación como Gradient Boosted Decision Tree, Redes Neuronales y Regresión Logística.

Por su parte, en [27] se presenta un sistema de recomendación para LoL en el que se identifican 14 clusters distintos de estilo de juego que puede tener un jugador y se generan recomendaciones de campeón específicas para cada tipo de estilo de juego. La idea para este proyecto es distinta: la recomendación debe hacerse para cualquier jugador, sin tener que saber información previa de él, es decir, basándose exclusivamente en los campeones seleccionados durante el Draft.

En general, a diferencia del LoL, existen muchos estudios relacionados con sistemas de recomendación de personajes para otros juegos, como es el caso de Dota 2 del que dispone de una gran cantidad de datos. Por ejemplo, para este juego en [28] se desarrolla un algoritmo que usa un predictor (con Regresión Logística y k-Nearest Neighbor) con una estrategia de búsqueda greedy. Este tipo de estrategias (greedy) requieren mucho tiempo y no podrían usarse para aplicaciones en tiempo real. Otro artículo similar para Dota 2 es el de Summerville et al. [29], en el que se usan redes neuronales LSTM (Long Short-Term Memory) y redes bayesianas para predecir el siguiente campeón que va ser seleccionado.

Como conclusión, se puede constatar que existe mucho interés por sistemas de recomendación en juegos con fases similares al Draft, y hay varias propuestas de ello en LoL. Sin embargo, no se ha encontrado ningún trabajo que trate el tema en profundidad, estudiando las distintas posibilidades para el predictor y que proponga un algoritmo final de recomendación. Por ello. podría tener interés un análisis como el que se explora en este proyecto.

3. VARIEDAD, EXTRACCIÓN Y DISTRIBUCIÓN DE LOS DATOS

3.1. Riot Games API

Como ya se comentó, League of Legends tiene una API (en inglés, Interfaz de Programación de Aplicaciones) que permite extraer diversos datos: datos generales del juego, datos sobre partidas, sobre jugadores, etc. La API de Riot Games fue una respuesta a jugadores y desarrolladores para proporcionales más recursos para contribuir con la experiencia de juego. Su importancia para este es proyecto es enorme, puesto que es de ahí de donde se obtienen los datos con los que se trabajará para diseñar el sistema de recomendación. Además, como la API está en continuo mantenimiento y evolución, siempre será posible obtener nuevos datos con los que actualizar el sistema de recomendación para cada versión o parche del juego.

Las peticiones a la API devuelven un mensaje con formato de texto JSON, que está formado por objetos que contienen parejas con la forma *"nombre": valor*, separados por comas y puestos entre llaves. Se trata de un formato fácil de leer y escribir para las personas, y muy fácil de analizar y generar para las máquinas, lo que lo convierte en un lenguaje ideal para el intercambio de datos.


```
{
    "profileIconId": 0,
    "name": "FNC Rekkles",
    "summonerLevel": 139,
    "accountId": 23989840,
    "id": 20717177,
    "revisionDate": 1533811016000
}
```


Figura 3.1: Ejemplo respuesta de la API en JSON

Para poder hacer peticiones es necesario disponer de una API key, que cada jugador tiene asociada a su cuenta y que puede solicitar de manera gratuita. Sin embargo, como es común en la mayoría de APIs, para minimizar el abuso y que haya estabilidad, las API keys tienen tasas límites de peticiones. Cada petición tendrá que incluir entonces el API key. Por ejemplo, la URL de la petición que devolvió la respuesta de la figura 3.1 fue:

https://euw1.api.riotgames.com/lol/summoner/v3/summoners/by-name/FNC%20Rekkles?api_key=<key>

Donde habría que reemplazar *<key>* por la correspondiente API key.

3.2. Datos a utilizar

Cuando en el estado del arte se comparaban los trabajos similares existentes con el presente proyecto, se argumentó que los únicos datos que se podían usar para elaborar un sistema de recomendación robusto en tiempo real (durante la selección de campeones), eran los propios campeones seleccionados. En realidad, con solo datos de los campeones seleccionados de cada partida no se podrían hacer más que estadísticas simples, es decir, no habría una capacidad de aprendizaje para dar respuesta a cada caso particular.

Sin embargo, la clave para poder usar técnicas de Machine Learning viene dada por la existencia de una serie de características que serán usadas como features en los distintos modelos. Estas características son, como ya se comentó, las estadísticas que tiene cada campeón (cierta cantidad de vida, cierta velocidad de movimiento, etc...). Aunque haya alguna estadística que pueda ser igual en dos campeones, ningún campeón tiene exactamente las mismas estadísticas que otro, lo que hará que para cada caso un campeón sea mejor que el resto.

Las estadísticas de cada campeón son fijas para cada parche, es decir, que en una misma versión del juego con hacer una única petición se pueden obtener todas las estadísticas y almacenarlas para utilizarlas posteriormente. Es importante resaltar que de una versión a otra estas estadísticas suelen cambiar, ya que los campeones suelen ser editados para intentar equilibrar el juego, por lo que los datos de partidas de una versión tienen que corresponder con los datos de estadísticas de esa propia versión. En la figura 3.2 se muestra la forma en la que se obtienen los datos de estadísticas a través de la API.

```
{1: {'attackdamageperlevel': 2.625, 'hpregen': 5.424, 'mpregen': 6.0, '
    spellblock': 30.0, 'armorperlevel': 4.0, 'critperlevel': 0.0, 'crit
    ': 0.0, 'movespeed': 335.0, 'name': 'Annie', 'spellblockperlevel':
    0.5, 'hpregenperlevel': 0.55, 'mp': 334.0, 'attackrange': 575.0, '
    hp': 511.68, 'attackdamage': 50.41, 'mpregenperlevel': 0.8, '
    attackspeedoffset': 0.08, 'armor': 19.22, 'hpperlevel': 76.0, '
    mpperlevel': 50.0, 'attackspeedperlevel': 1.36},
2: {'attackdamageperlevel': 3.5, 'hpregen': 8.512, 'mpregen': 7.466, '
    spellblock': 32.1, 'armorperlevel': 3.0, ...
                                ...
}
```

Figura 3.2: Datos de estadísticas de campeón proporcionados por la API

En total son 20 estadísticas (el nombre del campeón no es una estadística), de las cuales se usarán solo 18 ya que dos de ellas ("crit" y "critperlevel") son nulas en la mayor parte de los campeones. Se aprecia que, por ejemplo, el campeón representado por el número 1 (llamado Annie) tiene 19.22 de armadura o que el campeón 2 tiene 8.512 de regeneración de vida. Los algoritmos de predicción utilizarán los valores de las estadísticas de cada uno de los 10 campeones para cada partida y, por supuesto, el resultado de la partida (si gana el equipo azul o el equipo rojo).

De todos los datos que proporciona la API para cada partida, nos quedaremos simplemente con 11 números. El primer número indica el equipo ganador: si es 0 ganó el equipo azul y si es 1 ganó el equipo rojo. Los siguientes cinco números son los campeones del equipo azul y los cinco últimos son los campeones del equipo rojo. Además, como también se puede obtener el rol o posición de cada campeón, se ordenarán en ambos equipos siempre según la posición: Top, Jungle, Mid, ADC y Support. En la siguiente figura se muestra de forma esquemática como se guardarán los datos para cada partida.

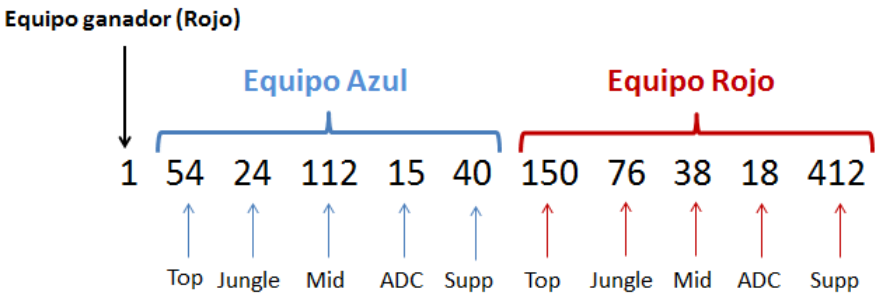


Figura 3.3: Esquema de los datos guardados por partida (ejemplo)

Es decir, en la Figura 3.1 se puede observar, por ejemplo, que en esa partida el campeón del equipo azul que va a la línea central es el 112, que el ADC del equipo rojo es el campeón 18 o que ha ganado la partida el equipo rojo. Representar una partida con tan solo un array de 11 enteros permite guardar una gran cantidad de datos sin ocupar mucho espacio y tener gran rapidez en el procesado. En la figura 3.4 se muestra un ejemplo del tipo de documento que contendrá los datos de las partidas,

```
0 516 107 268 110 98 78 121 112 498 12
0 14 113 268 498 3 420 35 163 18 12
1 8 59 61 498 201 27 24 4 429 497
0 54 60 69 236 12 41 154 13 429 412
1 14 121 7 498 98 6 59 55 222 497
1 14 64 69 18 44 41 164 163 51 223
```

Figura 3.4: Ejemplo del documento en el que se guardan los datos de las partidas

De esta forma, a la hora de entrenar el modelo de clasificación, si se utilizan n estadísticas por campeón, cada partida estará formada por un array con un total de $10 \cdot n$ datos numéricos, además del resultado de la partida (0 o 1). En los capítulos de predicción se desarrollará como escoger las estadísticas más importantes (selección de características o 'feature selection') y, en el caso de la predicción con Machine Learning Clásico, como transformar ese array complejo en otro que sea más comprensible para el modelo y que pueda proporcionar mayor precisión. Una vez se haya hecho este proceso los datos estarán listos para entrenar el modelo.

3.3. Extracción de los datos

Aunque existen librerías de ayuda para usar la API de Riot Games (como por ejemplo 'Cassiopeia' en Python), se ha preferido descartar su uso para tener un mayor control del código. El algoritmo explicado a continuación se ha desarrollado en Python, usando la librería 'Requests' para las peticiones HTTP.

Para entrenar los modelos de clasificación será necesario obtener una gran cantidad de datos, por lo que se ha desarrollado un método para obtener datos de manera automática a través de la API. La idea fundamental es obtener la ID de las partidas (*game_Id*) para poder hacer una petición a cada una de ellas y obtener los datos en el formato explicado en la sección anterior. Para obtener los *game_Id* se hará previamente una petición a la cuenta de un jugador (*acc_Id*) que devolverá una lista de las partidas que ha jugado.

En las propias peticiones sobre partidas se mostrarán las *acc_Id* del resto de jugadores que almacenaremos en una cola (Queue), lo que permite poder acceder a una gran cantidad de datos partiendo de una cuenta inicial (*seed_Id*). Además, en este proceso es bastante posible que se pueda solicitar información varias veces sobre una misma partida, ya que una cuenta se obtiene de una partida, y al obtener las partidas de esa cuenta se puede obtener la propia partida de la que se obtuvo. Para solucionarlo se almacenarán todas las *game_Id*, lo que permitirá no repetir datos y evitar hacer solicitudes innecesarias (puesto que, como se explicó, hay una tasa límite de peticiones).

En la figura 3.5 se presenta el diagrama de flujo del algoritmo usado para obtener los datos. Cuando se hace una petición para obtener la lista de partidas de una *acc_Id*, se debe seleccionar un intervalo de tiempo en el que se jugaron las partidas y un modo de juego. Ese intervalo es útil para poder seleccionar exclusivamente las partidas que correspondan a un parche o versión del juego. Para este proyecto se han obtenido partidas de la **versión 8.3.1** de LoL, que estuvo vigente desde el 7 al 22 de febrero de 2018. En esa versión había 139 campeones diferentes. Además, el modo de juego que se elige siempre es el de partidas clasificatorias que, como se comentó, son aquellas en las que los jugadores pueden ascender a través de ligas.

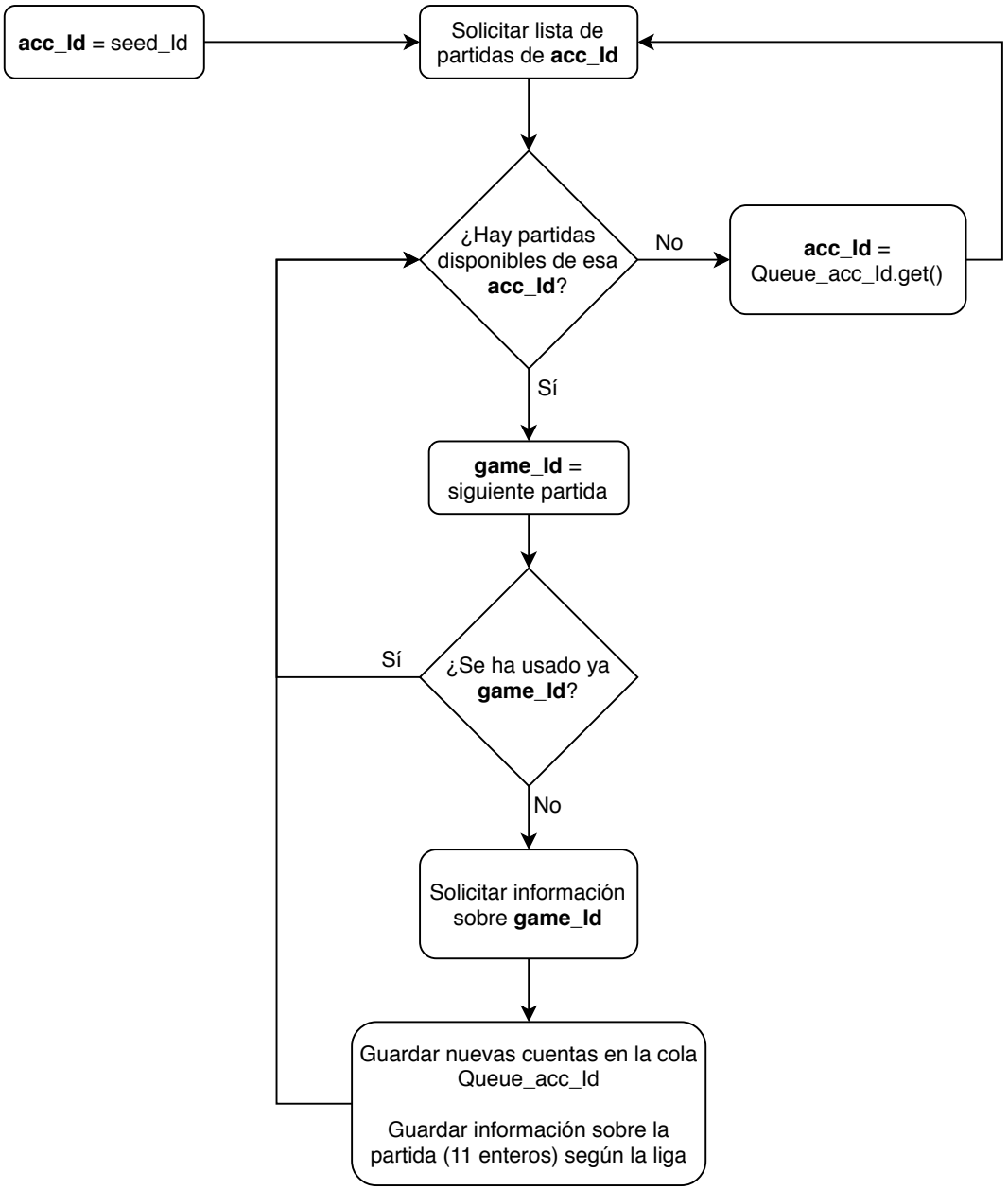


Figura 3.5: Diagrama de flujo para la obtención de datos de partidas

La cuenta inicial o semilla, *seed_Id*, se elige de manera aleatoria a través de los ficheros de Seed Data que proporciona la Riot Games en la página web de su API. Para las partidas de ligas más altas, donde hay menos jugadores y es más difícil obtener partidas, se obtienen directamente como semillas las cuentas de los mejores jugadores en la clasificación.

La lista de partidas obtenidas puede ser muy grande, por lo que se decide fijar obtener una número máximo de partidas por jugador (entre 5 y 15), y seleccionarlas de forma aleatoria entre todas las partidas de la lista. De la misma forma, la cola en la que se guardan las cuentas *Queue_acc_Id* se decide que tenga un tamaño máximo (alrededor de las 250 cuentas) por motivos de eficiencia en el funcionamiento del código, ya que se encolan muchas más cuentas de las que desencolan. De esta forma,

a medida que se va vaciando la cola se introducen nuevas cuentas sin tener que almacenar una gran cantidad de ellas.

Por último, es importante comentar por qué y cómo se hace la clasificación de una partida en una liga. La razón para clasificar las partidas en ligas es que el nivel entre los jugadores difiere mucho entre distintas ligas. La forma de jugar e incluso los tipos de campeones que se seleccionan varía mucho de una liga a otra. Por ello, para que la predicción del ganador pueda ser más precisa, es preferible que los datasets se separen en la distintas ligas.

En las peticiones de información con la API sobre una partida, el único dato que proporciona la API sobre la liga de un jugador, es la liga más alta a la que llegó cada jugador en la temporada anterior. Por lo tanto, es necesario crear algún método parar situar una partida dentro de una liga u otra.

Figura 3.6: Distribución de jugadores por ligas. Datos obtenidos de op.gg en agosto de 2018.

En una misma partida puede y suele haber jugadores que correspondan a distintas ligas, pero nuestra misión será hallar la liga media aproximada de la partida. Además, habrá que tener en cuenta que puede haber jugadores que no se hayan clasificado en ninguna liga en la pasada temporada. El método creado para clasificar partidas por ligas es muy sencillo: se aplica una serie de pesos a cada jugador dependiendo de la liga en la que acabó la temporada pasada y se hace la media aritmética. Para esta media solo se tienen en cuenta los jugadores que se clasificaron en la temporada pasada y si el número de jugadores sin clasificar es igual o mayor a 7, se descartará la partida por incertidumbre.

Una vez hallada la media aritmética, se clasifican en las distintas ligas según la Tabla 3.1. Como el número de partidas en Máster y Challenger es muy pequeño (solo hay un 0.04 % de los jugadores entre ambas), se ha decidido juntar ambas ligas en una sola denominada Master/Challenger o solo Challenger.

La idea detrás de los pesos asignados a cada una de las ligas viene dada por el hecho de que la diferencia de habilidad de los jugadores entre ligas se hace mayor a medida que se avanza por las ligas. Es decir, la diferencia de habilidad entre un jugador de plata y uno de oro no es tan grande como lo es entre un diamante y un master/challenger. De la misma forma, los intervalos escogidos para clasificar la partida se han elegido de forma muy restrictiva, de manera que si en una partida

TABLA 3.1: PESOS Y CLASIFICACIÓN POR LIGAS

Liga	**Peso**	**Clasificación partida**
Bronce	0	$media < 4$
Plata	5	$4 \leq media < 13$
Oro	15	$13 \leq media < 25$
Platino	30	$25 \leq media < 45$
Diamante	50	$45 \leq media < 70$
Master/Challenger	75/100	$media \geq 70$

hay mezcla de ligas, se incline hacia una liga inferior.

A modo de ejemplo, si en una partida hay cuatro oro, dos platino, dos plata y dos sin clasificar, la media aritmética de los pesos daría 16.25, por lo que la clasificaríamos como oro. Si, por ejemplo, en una partida hubiese seis oros y cuatro platas, se clasificaría como plata. Esto último es exactamente lo que se busca porque al haber tantos plata en la partida, significa que el nivel de la partida no llega a ser oro de verdad, y se prefiere asumir que se trata de un nivel de plata.

3.4. Distribución de los datos

Para elaborar el predictor del equipo ganador de la partida será necesario obtener una gran cantidad de datos y organizarlos de manera que se pueda diseñar la mejor arquitectura que proporcione la mayor precisión posible. Para ello, comúnmente en Machine Learning se divide el conjunto de datos completo en dos o tres partes. En este caso, como se busca ir modificando y mejorando el modelo, y hacer una evaluación final a un conjunto de datos que no haya sido usado previamente, se decide dividir el dataset (conjunto de datos) en tres partes:

- **Training set:** Usado para entrenar el modelo, es decir, para ajustar los parámetros del clasificador.
- **Validation set:** Para estimar como ha sido entrenado el modelo. Nos fijaremos sobre todo en la precisión de validación a la hora de elegir entre distintos modelos e incorporar mejoras.
- **Test set:** Usado exclusivamente para la evaluación final del modelo. Se reservará una parte del dataset para, al final del todo (una vez que se haya fijado el modelo final), hacer una última prueba de cómo responde el modelo a datos nuevos.

Se distingue entre un conjunto de validación y otro de prueba para comprobar que los cambios que se han ido aplicando al clasificador, que se basaban en la precisión

de validación, no se amoldan a los datos de validación. Haciendo una prueba final a un conjuntos de datos que no se haya utilizado servirá para saber si el modelo responde bien a datos nuevos, que es lo que desea al diseñar un clasificador.

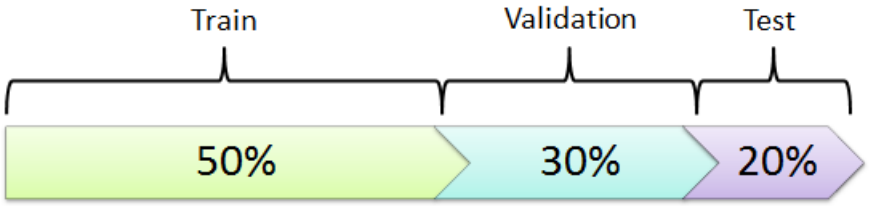


Figura 3.7: Distribución del dataset.

Tras un proceso obtención de datos de una duración aproximada de 90 horas, siguiendo el proceso explicado en la sección anterior, y usando cuatro API keys distintas, se ha obtenido una gran cantidad de partidas. La distribución del dataset se muestra en la Tabla 3.2. Se ha seguido la siguiente distribución de los datos en cada liga: 50 % de las partidas totales para el Training set, 30 % para el Validation set y 20 % para el Test set.

TABLA 3.2: DISTRIBUCIÓN DEL DATASET: NÚMERO DE PARTIDAS

	Train	**Validation**	**Test**	
Bronce	27610	16566	11044	**55220**
Plata	37601	22561	15041	**75203**
Oro	35280	21168	14111	**70559**
Platino	44550	26730	17820	**89100**
Diamante	22902	13741	9160	**45803**
Master/Challenger	607	363	242	**1212**
				337097

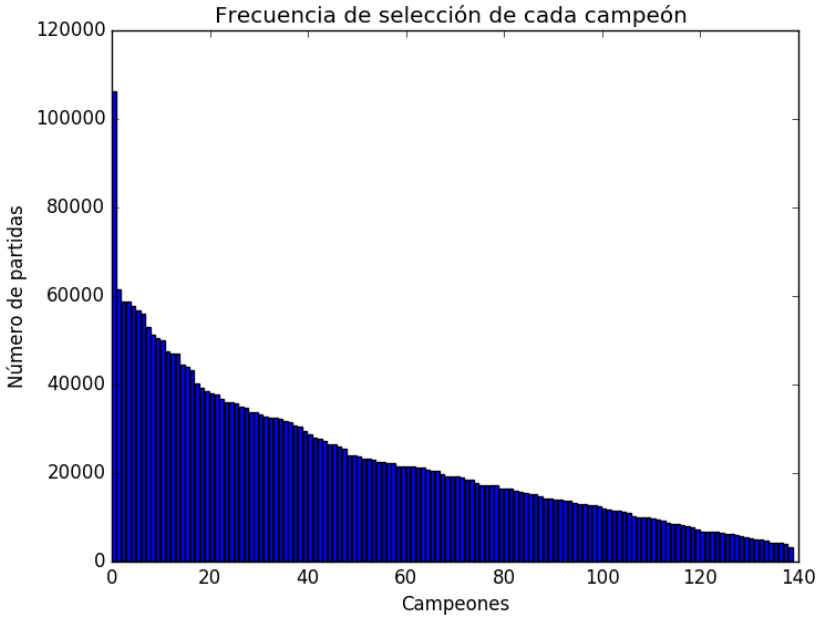
Se cuenta con un total de 337097 partidas entre todas las ligas. La razón para contar con tantos datos es que, como ya se ha comentado, el juego está pensado para que las partidas estén lo más igualadas posible. De esta manera, y aunque sea razonable que no se obtenga una alta precisión, se necesitará contar de todas formas con una gran cantidad de partidas para poder hacer un clasificador. Es por este motivo que la mitad del dataset (50 %) se utiliza para entrenar el modelo. Además, se deja un 20 % para los datos de test para tener suficiente confianza en la prueba final del modelo. El resto de los datos, que también necesitamos que sean suficientes, se dejan para los datos de validación.

Además, se puede observar que hay muchas menos partidas en Master/Challenger. Esto se debe a que hay una cantidad muy pequeña de jugadores en estas ligas

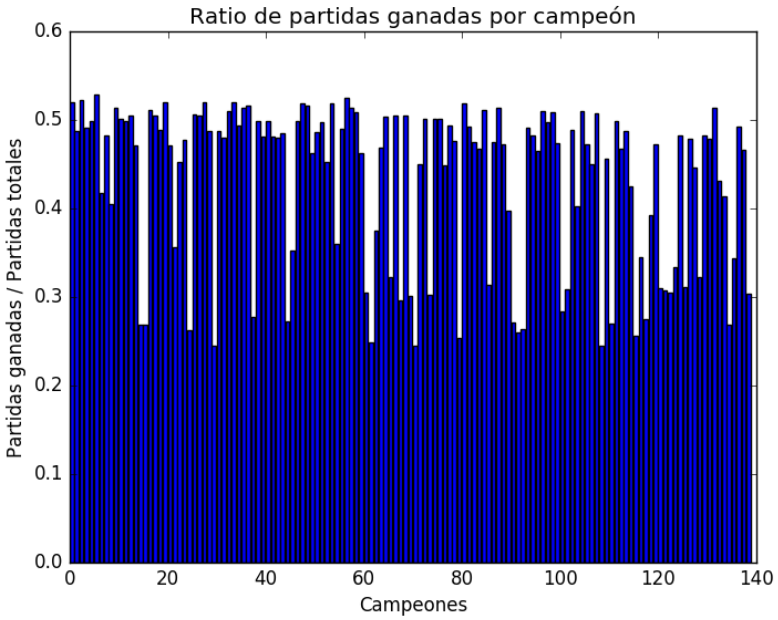
(ver Figura 3.3), por lo que se juegan muy pocas partidas y no se puede obtener una gran cantidad de datos para estas ligas.

3.4.1. Pre-análisis de los datos

Antes de empezar a diseñar el predictor, es conveniente hacer una breve visualización de los datos extraídos para hacerse una idea del problema a tratar.



(a) Frecuencia de selección de cada campeón



(b) Ratio de partidas ganadas por campeón

Figura 3.8: Frecuencia de selección y ratio de partidas ganadas

En la figura 3.8(a) se muestra el histograma con la frecuencia de selección de cada campeón, es decir, el número de veces que se ha seleccionado cada campeón en todas las partidas obtenidas. Además, se ha ordenado de mayor a menor frecuencia. Al haber un total de 337097 partidas y 10 campeones por partida, la suma de todas las barras da 3370970. Por su parte, la figura 3.8(b) muestra el porcentaje de veces que cada campeón ha ha ganado la partida. Es decir, se divide el número de partidas que ha ganado cada campeón entre el número de partidas totales en las que ha participado. Es importante tener en cuenta que el orden de campeones de la figura 3.8(b) es el mismo que el de la figura 3.8(a), para poder compararlos.

Se puede concluir que, pese haber una preferencia en la selección del campeón (ya que en la figura 3.8(a) se observa que unos campeones se utilizan más que otros), no hay ninguno que de garantía alguna de victoria. Esto puede dar a entender la dificultad que entraña el diseño de un predictor del ganador de una partida.

4. PREDICCIÓN CON MACHINE LEARNING CLÁSICO

4.1. Introducción y objetivos

En este capítulo se elaborará el predictor del equipo ganador de una partida completa (es decir, con los 10 campeones) siguiendo la estrategia llamada Machine Learning Clásico, que se refiere a aquellas ramas del Aprendizaje Automático que nos permitan hacer la predicción sin hacer uso de Deep Learning, que se tratará en el siguiente capítulo. La capacidad de predicción resultante será utilizada posteriormente para proporcionar una recomendación durante el Draft.

Para llevar a cabo la predicción tendremos que entrenar un modelo partiendo de $10 \cdot n$ datos por partida (donde n es el número de estadísticas de campeón seleccionadas) y el ganador de dicha partida (0 equipo azul, 1 equipo rojo). Como se ha mencionado en y será detallado en la siguiente sección, se puede considerar un problema de clasificación porque se intentará identificar a que categoría pertenece una partida, habiendo 2 posibles categorías: el ganador es el equipo azul o el ganador es el equipo rojo. Asimismo, es importante comentar que en todo momento se trata de Aprendizaje Supervisado ya que los algoritmos de los modelos aprenden a predecir la salida (output data, ganador de la partida) a partir de los datos de entrada (input data).

Sin embargo, surgen varios problemas que complican a priori el desarrollo del clasificador y que se pueden considerar como objetivos a desarrollar en este capítulo:

1. Los datos de estadísticas de campeón con los que contamos son muy variados y usar $10 \cdot n$ datos en el modelo puede no ser lo más productivo. Se buscará transformar los datos de forma que podamos facilitar al modelo la comprensión de estos y obtener una menor probabilidad de error a la vez que se reduce el tiempo de procesamiento. Este tipo de procedimiento se conoce en Machine Learning comunmente como “Feature Extraction”.

2. Se cuenta con 18 posibles estadísticas por campeón, lo que hace un total de 180 características (features) posibles para entrenar al modelo. No todas las características tendrán la misma importancia: unas serán de gran importancia para el resultado de la partida y otras no serán relevantes. Por ello, es necesario hacer una correcta selección de ellas (“Feature Selection”), lo que además reducirá el tiempo de entrenamiento del modelo al haber menos datos por partida.

3. Existen multitud de modelos de clasificación en Aprendizaje Supervisado: Logistic Regression, SVM, Nearest Neighbors, Decision Trees... Nuestra misión será estudiar los modelos más importantes, y evaluar como se ajustan a nuestro problema.

Antes de empezar a tratar estos problemas, se proporcionará una introducción teórica para entender como se realiza la clasificación en los modelos. Además, para empezar llevar a cabo los objetivos, será necesario contar un modelo de clasificación base sobre el que trabajar. Es decir, para poder hacer la selección de características (feature selection) y comprobar que el pre-procesado o transformación de los datos es efectivo, es necesario probarlo sobre un modelo de clasificación. Por ello, se decide que el modelo de clasificación inicial con el que realizar las pruebas sea Regresión Logística (Logistic Regression en inglés). Como se va a utilizar este algoritmo durante la mayor parte del capítulo, se considera conveniente explicar brevemente como funciona. Más tarde, cuando se hayan tratado los dos primeros problemas, podremos estudiar otros algoritmos de clasificación.

4.2. Teoría y herramientas de clasificación

Se ha explicado anteriormente que nuestro problema se trata de un problema de clasificación. En esta sección se hará una breve introducción teórica a este problema y se particularizará a un tipo de algoritmo de clasificación (la regresión logística). Si el lector ya estuviese familiarizado con esta materia, puede dirigirse directamente a la siguiente sección. Además, se explicarán también las herramientas que se usarán para crear los modelos de clasificación. La parte teórica de esta sección es un resumen adaptado de [30].

4.2.1. Problema de clasificación

En un problema genérico de clasificación se nos da un vector observación $\mathbf{x} \in \mathbb{R}^N$ que pertenece a una única clase o categoría, y, del set $\mathcal{Y}$. Dado que en nuestro caso solo existen dos posibles categorías, nuestro caso es un problema de clasificación binario, donde $\mathcal{Y} = \{0, 1\}$. El objetivo de clasificador es predecir el valor de y basándose en $\mathbf{x}$. Para diseñar el clasificador contamos con una serie de observaciones $\mathcal{S} = \{(\mathbf{x}^{(k)}, y^{(k)})\}_{k=1}^{K}$ donde, para cada observación $\mathbf{x}^{(k)}$, el valor de su categoría, $y^{(k)}$, es conocido.

Los algoritmos de clasificación se basan en dos hipótesis importantes:

- Todas las muestras del dataset son i.i.d (independientes e idénticamente distribuidas), es decir, que todas ellas son resultados independientes de una distribución desconocida $p(\mathbf{x}, y)$.

- Para cualquier dato de prueba, la unión formada por la muestra de entrada y su clase desconocida, $(\mathbf{x}, y)$, es un resultado de la misma distribución.

Estas dos suposiciones son esenciales para garantizar que un clasificador basado en $\mathcal{S}$ funcione bien cuando se aplica a nuevas muestras. Es importante notar que, a pesar de asumir la existencia de una distribución, dicha distribución se desconoce. De otro modo, podríamos ignorar $\mathcal{S}$ y aplicar teoría de decisión clásica para hallar el predictor óptimo basado en $p(\mathbf{x}, y)$. Dado que las pruebas iniciales las vamos a hacer usando un modelo de Regresión Logística, es conveniente explicar en qué consiste para entender como se hace el aprendizaje.

4.2.2. Regresión logística

La decisión clásica asume que cada cada muestra $(\mathbf{x}, y)$ es el resultado de un vector aleatorio $(\mathbf{X}, Y)$ con distribución conjunta $p_{\mathbf{X},Y}(\mathbf{x}, y)$. Denotaremos $\hat{y}$ como la salida del clasificador (o decisión), por lo que si $y = \hat{y}$ la decisión es un acierto, si no la decisión es un error. Un criterio natural de clasificación es elegir el predictor $\hat{Y} = f(\mathbf{x})$ de forma que la probabilidad de error, $P\{\hat{Y} \neq Y\}$, es mínima. Esto correspondería a la siguiente regla de decisión:

$$P_{Y|\mathbf{X}}(1|\mathbf{x}) \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} \frac{1}{2} \tag{4.1}$$

El clasificador que implementa esta regla de decisión se suele llamar *MAP* (Maximum A Posteriori). Sin embargo, MAP asume que $p_{\mathbf{X},Y}(\mathbf{x}, y)$ es conocida. Para nuestro caso, esto no serviría ya que tenemos que construir el clasificador con un dataset $\mathcal{S} = \{(\mathbf{x}^{(k)}, y^{(k)}) \in \mathbb{R}^N \times \mathcal{Y},\ k = 1, \ldots, K\}$. Muchos algoritmos de clasificación usan el dataset para obtener una estimación de las probabilidades de clase a posteriori y aplicarla para implementar una aproximación de la regla MAP.

La **regresión logística** asume que el etiquetado de clase binario $Y \in \{0, 1\}$ dada una observación $X \in \mathbb{R}^N$ satisface la expresión:

$$\begin{aligned} P_{Y|\mathbf{X}}(1|\mathbf{x}, \mathbf{w}) &= g(\mathbf{w}^\intercal \mathbf{x}) \\ P_{Y|,\mathbf{X}}(0|\mathbf{x}, \mathbf{w}) &= 1 - g(\mathbf{w}^\intercal \mathbf{x}) \end{aligned} \tag{4.2}$$

donde $\mathbf{w}$ es un vector de parámetros y $g(\cdot)$ es la *función logística*, que se define como:

$$g(t) = \frac{1}{1 + \exp(-t)} \tag{4.3}$$

Esta función se caracteriza por ser simétrica, monótona y tener valores contenidos en $0 \leq g(t) \leq 1$.

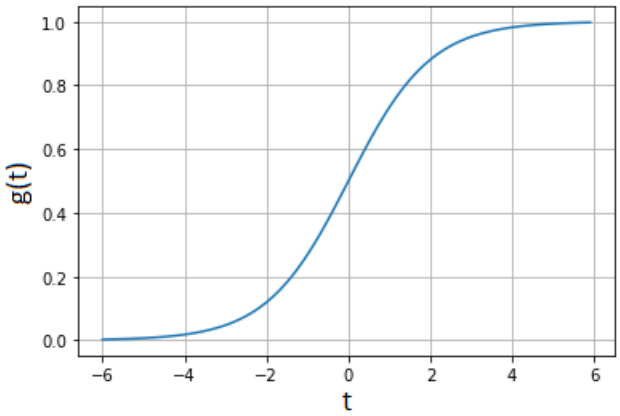


Figura 4.1: Función logística

El clasificador MAP basado en el modelo logístico tendrá la siguiente forma:

$$P_{Y|\mathbf{X}}(1|\mathbf{x},\mathbf{w}) = g(\mathbf{w}^\intercal\mathbf{x}) \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} \frac{1}{2} \tag{4.4}$$

Por simple inspección se observa que para que la función $g(t)$ sea mayor de 0.5, t tiene que ser mayor de 0. De la misma forma, para que $g(t)$ sea menor de 0.5, t tiene que ser menor de 0. Por consiguiente, los clasificadores basados en el modelo logístico tienen el origen, $\mathbf{x} = \mathbf{0}$, como la línea de decisión. Por tanto, el clasificador se nos queda como:

$$\mathbf{w}^\intercal\mathbf{x} \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} 0 \tag{4.5}$$

El modelo logístico puede ampliarse para construir clasificadores no lineales, usando transformaciones de datos no lineales. La forma general de un modelo de regresión logístico es:

$$P_{Y|\mathbf{X}}(1|\mathbf{x},\mathbf{w}) = g[\mathbf{w}^\intercal\mathbf{z}(\mathbf{x})] \tag{4.6}$$

donde $\mathbf{z}(\mathbf{x})$ es una transformación no lineal de las variables originales. El límite de decisión está dado en este caso por la ecuación:

$$\mathbf{w}^\intercal\mathbf{z} = 0 \tag{4.7}$$

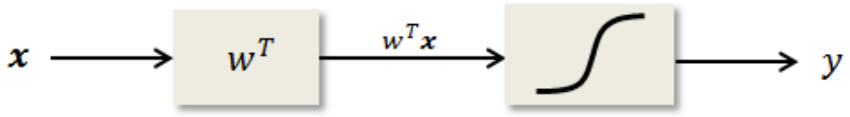


Figura 4.2: Diagrama de flujo de regresión logística

El objetivo ahora es estimar el vector de parámetros $\mathbf{w}$ de acuerdo a cierto criterio y usando el dataset $\mathcal{S}$. Este proceso, que se denomina inferencia, permitirá usar el vector estimado $\hat{\mathbf{w}}$ para clasificar una nueva observación como:

$$\hat{y} = \arg\max_{y} P_{Y|\mathbf{X}}(y|\mathbf{x}, \hat{\mathbf{w}}). \tag{4.8}$$

Dos criterios para optimizar la selección del vector de parámetros son:

- Maximum Likelihood (ML):

$$\hat{\mathbf{w}}_{\text{ML}} = \arg\max_{\mathbf{w}} P_{\mathcal{S}|\mathbf{W}}(\mathcal{S}|\mathbf{w}) \tag{4.9}$$

- Maximum A Posteriori (MAP):

$$\hat{\mathbf{w}}_{\text{MAP}} = \arg\max_{\mathbf{w}} p_{\mathbf{W}|\mathcal{S}}(\mathbf{w}|\mathcal{S}) \tag{4.10}$$

ML se basa en minimizar la función log-likelihood negativa. Al hallar el gradiente de esta función se obtendrá una función de la que no se podrá despejar el vector de parámetros $\hat{\mathbf{w}}_{\text{ML}}$, por lo que se tendrá que usar un algoritmo iterativo para buscar el mínimo. Por ejemplo, *Gradient descent* es un algoritmo iterativo de optimización de primer orden que permite obtener un simple algoritmo con el que hallar el vector de parámetros.

Por su parte, el criterio MAP asume que los pesos de la regresión logística (los valores de $\mathbf{w}$) siguen una distribución de probabilidad a priori $p_{\mathbf{W}}(\mathbf{w})$. De esta forma, aplicando la regla de Bayes a la densidad a posteriori $p_{\mathbf{W}}(\mathbf{w})$, se puede obtener de (4.10) la estimación MAP:

$$\hat{\mathbf{w}}_{\text{MAP}} = \arg\min_{\mathbf{w}} \{L(\mathbf{w}) - \log [p_{\mathbf{W}}(\mathbf{w})]\} \tag{4.11}$$

donde $L(\cdot)$ es la función log-likelihood negativa. Para hallar $\hat{\mathbf{w}}_{\text{MAP}}$ se escoge una distribución para $\mathbf{W}$. Según la distribución escogida, se distingue entre regularización L1 (Laplaciana) y regularización L2 (Gaussiana). En todas las pruebas se va a *utilizar la regularización L2*, que es la que viene por defecto en el algoritmo que se usará. Esta regularización asume que $\mathbf{W}$ es una variable Gaussiana de media cero y matriz de varianzas $v\mathbf{I}$,

$$p_{\mathbf{W}}(\mathbf{w}) = \frac{1}{(2\pi v)^{N/2}} \exp\left(-\frac{1}{2v}\|\mathbf{w}\|^2\right) \tag{4.12}$$

por lo que el estimador MAP se convierte en:

$$\hat{\mathbf{w}}_{\text{MAP}} = \arg\min_{\mathbf{w}} \left\{L(\mathbf{w}) + \frac{1}{C}\|\mathbf{w}\|^2\right\} \tag{4.13}$$

El término adicional introducido en el algoritmo de optimización se suele llamar término de regularización. Particularmente, el parámetro C (en este caso $C = 2v$) se llama *inverse regularization strength*. Regularizar es aplicar una penalización para aumentar la magnitud de los valores de los parámetros y así reducir el sobreajuste de los parámetros (overfitting). Por ello, más adelante será importante seleccionar un valor correcto de este parámetro para mejorar la precisión en la clasificación.

4.2.3. Herramientas de clasificación - Scikit Learn

Aunque es importante entender el funcionamiento de los algoritmos de clasificación, el objetivo de este proyecto no es programar algoritmos propios. Hay librerías implementadas de manera muy eficiente como es el caso de *Scikit-Learn*[1] para Python, que es la herramienta que utilizaremos en la parte de Machile Learning Clásico.

Scikit-Learn es una de las librerías más populares de algoritmos de Machine Learning en Python. Tiene buena documentación, es fácil de usar y proporciona un gran rendimiento. Además, usaremos *pandas*, una librería de estructuras de datos optimizada y compatible con Scikit-Learn, que servirá para introducir las muestras al clasificador.

En los modelos de clasificación hay dos procesos o métodos fundamentales:

1. *Fit method*: donde se recibe los datos de entrenamiento y el modelo aprende (por ejemplo, en Regresión Logística se hallan los valores del vector de parámetros con un algoritmo de iteración).

2. *Predict method*: usa los parámetros estimados en el proceso anterior para recibir una serie de muestras y devolver las predicción de clase (que será aquella clase que obtenga mayor probabilidad para cada input).

En la figura 4.3 se incluye el código de un ejemplo simple que muestra como se haría el entrenamiento de un modelo de regresión logística y la predicción de nuevas muestras.

El vector *y_test* de la figura 4.3 contendrá las clases que ha predicho el modelo para las muestras de *y_train*. Si se contase con las clases reales de *y_train* y se tratase de validar como funciona el modelo, simplemente habría que usar una función de precisión (por ejemplo, en Scikit-Learn existe *metrics.accuracy*()), que simplemente calcula el porcentaje de partidas que han sido predichas correctamente.

[1] `http://scikit-learn.org/stable/`

```
# Importamos pandas y Scikit-Learn
import pandas as pd
from sklearn import linear_model # Linear Model contiene el modelo de
    Logistic Regression

# Transformamos los datos para mayor eficiencia
X_train = pd.DataFrame(data_X_train)
X_test = pd.DataFrame(data_X_test)
y_train = data_y_train # Es un vector con las correspondientes clases
    de X_train

# Creamos el modelo de Logistic Regression y lo entrenamos
model = linear_model.LogisticRegression()
model.fit(X_train, y_train)

# Usamos el modelo para predecir nuevas clases
y_test = model.predict(X_test)
```

Figura 4.3: Funcionamiento de un modelo de clasificación en Scikit-Learn

4.3. Modelo principal

El modelo básico principal, a partir del cual se harán las distintas pruebas y que sirve como base para comenzar a estudiar todas las posibles mejoras, tiene las siguientes características:

- Se emplea un algoritmo de Regresión Logística con validación cruzada para la selección del parámetro C (inverse regularizaton strength, explicado en la parte teórica). La validación cruzada o cross-validation (CV) es una técnica muy utilizada en Machine Learning que consiste en calcular la media aritmética obtenida de las medidas de precisión de validación sobre diferentes particiones del dataset. De esta forma se asegura que los resultados son independientes de la partición entre datos de entrenamiento y validación.
- Se utilizan todas las estadísticas de campeón posibles. En total son 18 estadísticas de las 20 que se mostraron en la sección 3.2 (se han descartado 'crit' y 'critperlevel' por ser nulas en prácticamente todos los campeones). Por tanto cada partida estará formada por un vector de 180 datos (características).
- No se aplica ningún proceso a esos datos (ninguna transformación, normalización, selección de características...) ya que el objetivo es partir de una precisión inicial a partir de la cual se puedan decidir si una mejora tiene éxito o no.

El modelo se construye y se entrena de manera similar a la figura 4.3. Es importante recordar que el modelo se entrena siempre con los datos de entrenamiento

(training set) y se prueba su eficacia con el validation set (el test set, como se explicó, solo se utilizará para probar el modelo definitivo al final del capítulo). Aunque nos fijaremos sobre todo en la precisión de validación, también es conveniente mostrar en ciertas ocasiones la precisión de entrenamiento (para ver como responde el modelo entrenado a las propias partidas que lo han entrenado y averiguar si está habiendo sobreajuste, es decir, si el modelo se ajustando demasiado a los datos de entrenamiento y no es capaz de predecir otras muestras nuevas).

TABLA 4.1: PRECISIÓN DEL MODELO PRINCIPAL

Liga	Train Accuracy	Validation Accuracy
Bronce	0.5533	0.5383
Plata	0.5431	0.5326
Oro	0.5410	0.5245
Platino	0.5328	0.5235
Diamante	0.5441	0.5280
Challenger	0.6322	0.5152

En la tabla 4.1 se observan las precisiones para todas las ligas. Como se comentó en los objetivos del proyecto, las precisiones esperadas para el clasificador son bajas (ya que los propios desarrolladores del juego buscan que las partidas estén lo más equilibradas posible). Aun así, sorprende que en ligas como la de bronce se aproxime a un 54 % con un modelo tan básico.

Como se puede observar, la precisión para la liga de Master/Challenger es más baja que el resto. Esto se puede explicar por la pequeña cantidad de partidas que tenemos de esta liga (ver tabla 3.2) que tiene como consecuencia que el modelo se ajuste demasiado a las partidas de entrenamiento y no haga una buena predicción para las partidas de validación.

4.4. Pre-procesado de los datos

En esta sección se tratarán los dos primeros problemas que encontrábamos al diseñar el modelo de clasificación: facilitar la comprensión de los datos al modelo y reducir el número de características. Estos problemas están dentro de una fase muy típica dentro del Aprendizaje Autómatico que es el "pre-procesado de los datos" (Data Preprocessing). Podemos distinguir varios beneficios como resultado de llevar el preprocesado de los datos [31]:

- **Reducir el sobreajuste**: cuanto menos redundante sean los datos menor oportunidad habrá de hacer decisiones basadas en ruido.

- **Mejorar la precisión**: menor cantidad de datos confusos mejorará la precisión en la predicción.
- **Reducir el tiempo de entrenamiento**: los algoritmos funcionan más rápido cuantos menos datos haya.

Distinguiremos entre tres fases importantes que se desarrollarán en esta sección: *normalización*, transformación de datos (que puede considerarse como *Feature Extraction* ya que se construye un nuevo set de características a partir del set original) y *Feature Selection* (Selección de Características).

Con los datos con los que contamos, fases comunes del pre-procesado de datos como puedan ser limpieza de datos o la eliminación de outliers (valores atípicos) no son necesarias porque los valores de los datos están contenidos en intervalos fijos (ya que las estadísticas de los campeones son fijas para cada campeón).

4.4.1. Normalización

La normalización de los datos es un paso muy común del pre-procesado de datos en Machine Learning. El objetivo de la normalización es obtener un dataset en el que todos las coordenadas de entrada tengan una escala similar. La razón para normalizar es que los algoritmos de aprendizaje suelen mostrar menos problemas de inestabilidad y convergencia cuando los datos están normalizados.

Hay varios métodos para normalizar. El más común es aquel que produce una media y varianza unidad en los datos. Para ello simplemente habría que calcular la media μ y desviación estándar σ de manera independiente en cada característica (feature) y hacer la transformación:

$$X_{new} = \frac{(X - \mu)}{\sigma} \tag{4.14}$$

Scikit-Learn cuenta con distintos algoritmos de cambio de escala. A continuación se explica cada uno de forma breve:

- **StandardScaler**: sigue el método comentado, haciendo uso de (4.14)
- **MinMaxScaler**: escala los datos de manera que los valores de cada feature estén en el rango $[0, 1]$. Para ello simplemente se hayan el valor máximo y mínimo de cada feature y se hace la tranformación:

$$X_{new} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{4.15}$$

- **MaxAbsScaler**: escala cada feature con su valor máximo absoluto, de manera que los valores absolutos siempre están en el rango $[0, 1]$ (pero a diferencia del anterior, en este caso sí hay valores negativos)

- **RobustScaler**: a diferencia de los anteriores, este está basado en percentiles, lo que hace que sea robusto frente a outliers (de ahí su nombre).
- **QuantileTransformer**: aplica una transformación no lineal de manera que la función de densidad de probabilidad de cada feature se asignará a una distribución. Hay dos distribuciones posibles: uniforme y gaussiana.

StandarScaler, MinMaxScaler, MaxAbsScaler funcionan mal con outliers. Por su parte, RobustScaler y QuantileTransformer son robustas a outliers, pero eso implica que colapsarán automáticamente cualquier outlier a los límites de rango (0 y 1). Por la propia naturaleza de los datos que se manejan en este proyecto, como se comentó anteriormente los outliers no serán un problema.

Hay dos condiciones importantes que se tienen que tener en cuenta a la hora de normalizar un dataset:

1. La misma transformación debe ser aplicada a cada parte del dataset (train, validation y test). Es decir, por ejemplo en StandarScaler la misma media y varianza debe ser usada para los distintos sets. Para ello, lo más común es hallar los parámetros de la transformación usando el train set y aplicarlos al resto de sets.
2. La normalización es un proceso que siempre va al final de todo el pre-procesado de los datos, es decir, justo antes de introducir el modelo. No es lógico normalizar los datos y a continuación hacer algún cambio más en ellos.

En la tabla 4.2 se muestran la distintas precisiones de validación usando los distintos algoritmos de normalización de Scikit-Learn. El modelo usado es el mismo que el modelo principal, pero normalizando los datos antes de introducirlos al algoritmo. Para QuantileTransformer se indican los resultados con distribución uniforme ya que con distribución normal los resultados fueron peores.

TABLA 4.2: PRECISIÓN DE VALIDACIÓN POR ALGORITMO DE NORMALIZACIÓN (REGRESIÓN LOGÍSTICA)

Liga	Sin Normalizar	Standard	MinMax	MaxAbs	Robust	Quantile
Bronce	0.5383	0.5393	0.5399	0.5404	0.5413	**0.5414**
Plata	0.5326	0.5348	0.5347	**0.5351**	0.5343	0.5325
Oro	0.5245	0.5333	0.5329	0.5337	**0.5341**	0.5335
Platino	0.5235	0.5335	**0.5340**	0.5326	0.5329	0.5321
Diamante	0.5279	0.5270	0.5270	**0.5281**	0.5258	0.5289
Challenger	0.5151	0.5179	0.4848	0.5234	**0.5289**	0.4848

Normalizando los datos se obtienen buenos resultados, llegando a aumentar más del 1 % la precisión en algunas ligas (que aunque parezca poco, supone un gran aumento para el difícil problema de clasificación que se trata). Como era de esperar, todos los algoritmos de normalización obtienen resultados similares. Aunque se podría usar cualquiera, se decide *utilizar MaxAbs como algoritmo de normalización* durante el resto de elaboración del modelo en este capítulo.

4.4.2. Feature extraction

Con el modelo principal se están usando 180 datos por partida en el modelo. Se desea estudiar si se mantiene o mejora la precision del modelo principal haciendo uso de alguna transformación que reduzca el número de datos por partida. El objetivo de transformar los datos sería reducir la dimensionalidad del problema: si se consigue reducir las dimensiones de los datos a una cantidad inferior sin perder información relevante para la clasificación, es posible que el aprendizaje funcione mejor, en el sentido de que habrá mejor correspondencia entre las prestaciones en entramiento y validación. Nuestra estrategia parte del hecho de que existen dos equipos diferentes y en cada equipo hay cinco campeones diferentes.

Surgen varias posibles estrategias que, sin tener en cuenta la posición de los campeones (top, mid, etc.), pueden ayudar a mejorar la precisión del modelo. Se distingue entre dos problemas y se aportan soluciones:

- *Primer problema*: Cada equipo está formado por sus propios 5 campeones.
  - **Solución 1**: sumar entre sí las estadísticas comunes de los 5 campeones en cada equipo. En otras palabras, en vez de tener 5 valores distintos de la misma estadística en un equipo (una para cada jugador), que cada estadística en un equipo esté representada por un solo valor. De esta forma pasará de haber $10 \cdot n$ datos por partida a $2 \cdot n$ datos, siendo n el número de estadísticas de campeón usadas.
  - **Solución 2**: multiplicar entre sí las estadísticas comunes de los 5 campeones de cada equipo y hacer la raíz quinta de este resultado (para evitar números demasiado grandes o pequeños). De la misma forma, cada partida será representada por $2 \cdot n$.
- *Segundo problema*: Existen dos equipos distintos. Para solucionar este problema hay que haber implementado alguna de las dos soluciones del primer problema.
  - **Solución A**: restar las estadísticas comunes entre ambos equipos. Es decir, una vez se ha seguido alguna solución para el primer problema, hacemos que se represente un solo valor por estadística que será el resultado de restar el valor de un equipo y el valor del otro. Esto hará que una

partida se represente con tan solo n datos, que corresponderá al número de estadísticas seleccionadas.

- **Solución B**: dividir las estadísticas comunes entre ambos equipos. Cada partida se representaría finalmente por n valores.

Las soluciones 2 y B plantean un problema: que se manejen valores nulos. Por ejemplo, si una estadística de un campeón es 0, en la multiplicación este valor se arrastrará aunque el resto de estadísticas puedan no ser 0. De la misma forma, si una estadística de un equipo vale 0, la división entre ambas no sería posible. Por ello, se decide implementar solo en las estadísticas que puedan ser nulas la otra solución equivalente. Es decir, por ejemplo, cuando se use la solución 2, solo en las estadísticas que puedan tener valores nulos se usará la solución 1.

Se tratará de averiguar como responde cada una de las posibles soluciones (1, 2, 1_A, 1_B, 2_A o 2_B). A modo de ejemplo, en la figura 4.4 se muestra de forma gráfica como se haría la transformación de los datos de una partida para la solución 1_A.

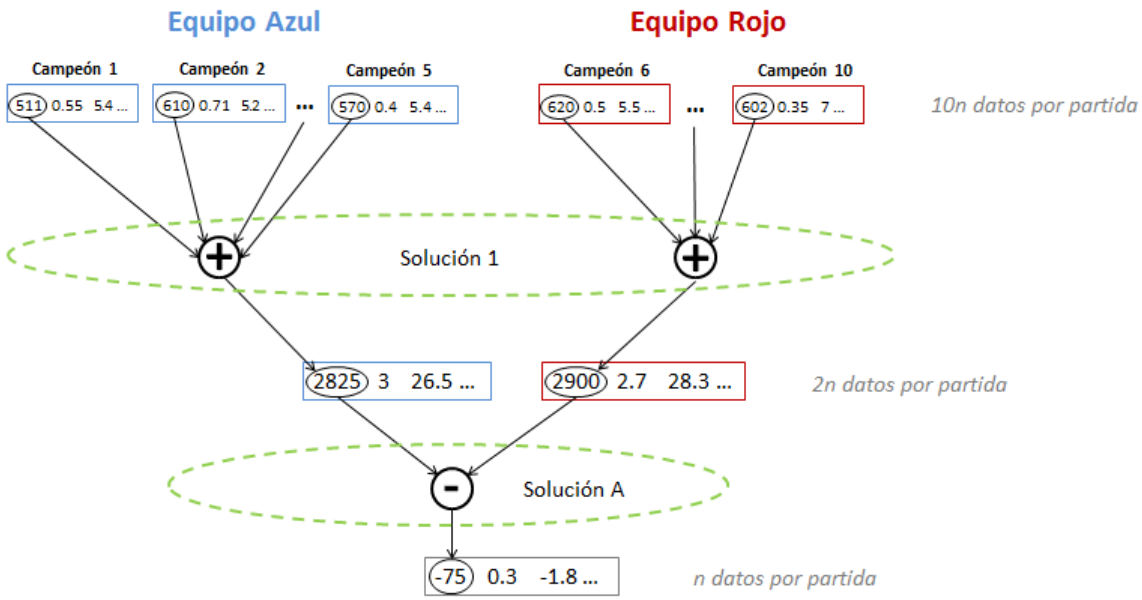


Figura 4.4: Esquema de la transformación de datos usando la transformación 1_A

Para las pruebas de las distintas transformaciones se sigue empleando Regresión Logística con validación cruzada para la selección del parámetro C. En este caso se normalizan los datos con *MaxAbsScaler*. Los resultados de cada solución para cada liga se muestran en la tabla 4.3.

TABLA 4.3: PRECISIONES DE VALIDACIÓN POR TRANSFORMACIONES

Liga	Original	1	2	1_A	1_B	2_A	2_B
Bronce	**0.5383**	0.5261	0.5233	0.5282	0.5253	0.5208	0.5222
Plata	**0.5326**	0.5201	0.5191	0.5243	0.5138	0.5238	0.5234
Oro	0.5245	0.5210	0.5221	0.5225	0.5241	0.5241	**0.5253**
Platino	**0.5235**	0.5189	0.5128	0.5180	0.5176	0.5149	0.5147
Diamante	**0.5280**	0.5176	0.5144	0.5220	0.5190	0.5202	0.5203
Challenger	0.5152	**0.5344**	0.5179	0.4848	0.4848	0.5262	0.4848

Como se puede observar, ninguna de las transformaciones ha tenido éxito. En la liga de Challenger, algunas transformaciones obtienen mayor precisión que el modelo básico principal, pero como hay un número tan pequeño de partidas de validación para esa liga no se tiene en cuenta. Se decide continuar con el modelo inicial.

Es importante comentar que se valoró el uso de hacer transformaciones con Polynomial Features de Scikit Learn. Lo que hace esta función es generar una nueva matriz de características formada por combinaciones polinómicas de las características con un grado específico. El problema es que al contar con 180 características (sin hacer selección de características), la matriz resultante sería enorme. Por ejemplo, para una transformación polinómica de grado 2 (la menor posible), cada partida tendría un total de 16471 características, que sería inviable por la lentitud a la hora de entrenar y predecir las muestras.

4.4.3. Feature Selection

La selección de características es un proceso importante de cara a mejorar la precisión y el rendimiento en problemas de datasets con muchas dimensiones. Se trata de seleccionar automáticamente aquellas características que más contribuyen a la predicción.

En esta sección se explorarán varias estrategias de selección de características [31] con el fin de intentar escoger aquellas estadísticas que verdaderamente tengan importancia para decidir el equipo ganador de la partida.

Univariate Feature Selection

Funciona empleando pruebas estadísticas para seleccionar aquellas características que tienen la mayor relación con la salida. Es decir, independientemente del algoritmo de clasificación que se use a continuación, este método relaciona únicamente entradas con salidas.

Scikit-Learn proporciona la función *SelectKBest* que selecciona las k features con

mayor puntuación. Esa puntuación puede obtenerse a través de diferentes pruebas estadísticas. Aunque existen otras, las pruebas estadísticas que proporciona la propia librería y que más se ajustan para clasificación son : ANOVA F-test, Chi-squared test y Mutual Information. Chi-squared no puede ser usado en nuestro modelo porque requiere que los valores de entrenamiento no sean negativos (y alguna estadística de campeón como "attackspeedoffset" puede tener valor negativo).

A continuación se describen brevemente las funciones de puntuación de Scikit-Learn que se probarán con *SelectKBest*:

- *f_classif*: se calcula el ANOVA F-Test para las muestras introducidas. El F-test en ANOVA (Analysis of Variance) se usa para evaluar si los valores esperados de una variable cuantitativa dentro de varios grupos predefinidos difieren entre sí.
- *mutual_info_classif*: se estima la información mutua para una variable objetivo discreta. La información mutua entre dos variables aleatorias es un valor no negativo que mide la dependencia entre variables. Cuanto mayor es su valor, mayor dependencia tienen.

Al final de la sección se probarán ambos algoritmos junto al resto de métodos de selección de características.

Recursive Feature Elimination (RFE)

RFE funciona eliminando de forma recursiva las características y construyendo un modelo en aquellas características que se mantienen. Es decir, dado un estimador externo (en este caso, de regresión logística) que asigna una serie de pesos a las características, se van seleccionando las features a medida que, de forma recursiva, se van considerando menores colecciones de características. El funcionamiento es el siguiente:

1. El estimador se entrena con un set inicial de features.
2. La importancia inicial de cada feature se obtiene con los atributos obtenidos del modelo (*coef_* ó *feature_importances_*)
3. Las features menos importantes se extraen del set de features.
4. Se repite el proceso de manera recursiva hasta llegar al número de features deseado

Para el problema de predicción del equipo ganador, el número de features no es importante. Se seleccionarán las estadísticas que den la mayor precisión, independientemente del número que sean.

Principal Component Analysis (PCA)

PCA usa Descomposición en Valores Singulares (SVD) para hacer una reducción de la dimensionalidad lineal, es decir, proyectar los datos en espacio dimensional menor. Aunque no selecciona características como tal, se puede elegir el número de dimensiones o componentes principales a obtener.

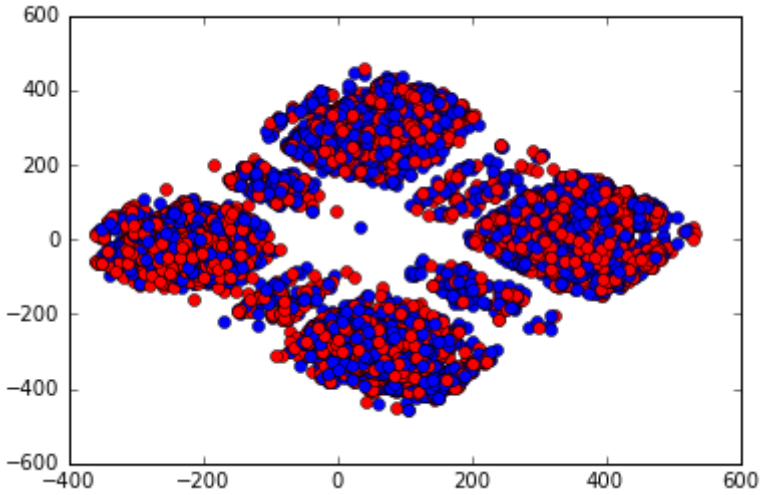


Figura 4.5: Representación de PCA en 2 componentes de los datos de entrenamiento de la liga Bronce.

Resulta muy interesante ver una representación del PCA en dos dimensiones del problema. De esta forma se puede llegar a visualizar la relación entre los datos de entrada y de salida, algo que no es tan sencillo de hacer por la gran cantidad de características que hay. En la figura 4.5 se muestra la representación de dos componentes del PCA de los datos de entrenamiento de la liga de Bronce, sin normalizar.

Se puede apreciar que no hay una relación clara entre las dos clases ("gana el equipo azul" o "gana el equipo rojo") ya que los puntos están dispersos sin seguir ninguna distribución clara. Esto puede explicar que no se obtengan precisiones altas en los clasificadores.

Feature Importance

Los modelos que usan métodos combinados (que se estudiarán en 4.5.5) como Random Forest o Gradient Boosting pueden hallar también la importancia relativa de cada característica. Estos valores de importancia pueden ser usados posteriormente para hacer un proceso de selección.

El proceso para obtenerlos es sencillo: se entrena el modelo ("fit") con los datos y se obtiene el atributo del modelo llamado *feature_importances_*. Este devuelva una lista que corresponde a la importancia de cada feature. La suma de todas las

importancias da la unidad, por lo que se pueden hacer comparaciones entre las distintas características para hacer la selección.

Comparación de los distintos métodos

Comparar todos los métodos de selección de características es un trabajo complicado, ya que no todos ellos seleccionarán las mismas características como las mejores. Además, en la mayor parte de los métodos hay que seleccionar el número de features que se quieren obtener, sin saber cuál es el número exacto que proporciona la mayor precisión. Por ello, es necesario realizar un gran número de pruebas para hacer una correcta selección con cada método.

En todas las pruebas para todos los métodos de selección de características se usan los datos de entrenamientos normalizados con *MaxAbsScaler*. *RFE* se hace con Logistic Regression y *Feature Importance* con Gradient Boosting.

Se llegan a las siguientes conclusiones:

1. Al hacer una proyección de las características a un cierto número componentes, *PCA* no permite una puntuación de cada característica por separado, por lo que es difícil compararlo junto al resto de métodos. A pesar de ello, *PCA* no mejora ni se acerca a la precisión obtenida previamente (no supera el 52 % en ninguna liga, para ningún número de dimensiones).

2. Con ambos métodos de Univariate Feature Selection (*f_classif* y *Mutual Information)* se obtienen resultados parecidos (tabla 4.4, pero *Mutual Information* requiere más tiempo de procesamiento, por lo que se prefiere el uso de *f_classif*.

3. *RFE* obtiene resultados similares a los métodos de Univariate Feature Selection, pero requiere más tiempo incluso que *Mutual Information*. Esto es así porque nuestro dataset contiene una gran cantidad de características (180), y en cada paso del algoritmo de recursión es necesario crear un nuevo modelo y entrenarlo, lo que lleva muchísimo tiempo. Por ello se descarta el uso de *RFE*.

4. *Feature Importance* también da resultados similares a los anteriores. El principal problema es que se tiene que entrenar otro modelo diferente para poder hallar las puntuaciones y si no han hecho pruebas previas no se puede saber si el modelo es fiable.

Teniendo todo esto en cuenta se decide finalmente usar *SelectKBest* con función de puntuación *f_classif* (aunque queda abierta la posibilidad de usar *Feature Importance*). En la tabla 4.4 se muestran las precisiones de validación por cada liga, tras entrenar el mismo modelo principal normalizado con *MaxAbsScaler* pero con distinto número de características seleccionadas (K).

TABLA 4.4: PRECISIONES DE VALIDACIÓN PARA DISTINTO NÚMERO DE CARACTERÍSTICAS SELECCIONADAS CON SELECTKBEST (F_CLASSIF)

Liga	K=180	K=175	K=170	K=160	K=150	K=140	K=130	K=120
Bronce	0.5404	0.5398	**0.5417**	0.5398	0.5374	0.5385	0.5359	0.5308
Plata	0.5351	**0.5361**	0.5352	0.5339	0.5347	0.5351	0.5296	0.5303
Oro	0.5337	0.5337	**0.5342**	0.5319	0.5330	0.5311	0.5299	0.5276
Platino	0.5326	**0.5331**	0.5320	0.5287	0.5289	0.5270	0.5270	0.5236
Diamante	0.5281	0.5275	0.5265	0.5269	0.5249	**0.5283**	0.5235	0.5244
Challenger	0.5234	0.5234	0.5179	**0.5289**	0.5234	0.5151	0.5179	0.5132

Aunque para ciertos valores de K la precisión aumente ligeramente respecto al número original (K = 180), se aprecia cómo en general a medida que se reduce el número de características la precisión decrece. Sin embargo, la conclusión importante que podemos extraer de aquí es que podemos llegar a reducir en gran parte el número de características sin llegar a perder demasiada precisión, consiguiendo una mayor velocidad en el procesamiento de lo datos (que puede ser muy útil en el sistema de recomendación).

4.5. Selección de modelo

Hasta ahora se ha utilizado un único algoritmo de clasificación, Logistic Regression. El objetivo en esta última sección es estudiar varios clasificadores y evaluar como responden al problema planteado.

Existe una gran cantidad de algoritmos de clasificación en Aprendizaje Supervisado. Como no es posible tratar todo ellos, se tratará de analizar al menos un algoritmo de cada clase o grupo. A continuación se muestran los grupos más importantes y el algoritmo de clasificación de Scikit-Learn que se estudiará en concreto de cada uno:

- **Modelos Lineales**: es el grupo al que pertenece la Regresión Logística.
- **Máquinas de vectores soporte**: los clasificadores SVM (Support Vector Machines) son muy usados por su simplicidad y buenos resultados. Se tratará el algoritmo fundamental de clasificación, *SVC* (Support Vector Classification)
- **Vecinos más Próximos**: se estudiará el algoritmo *KNeighborsClassifier*, que es el más común para clasificación
- **Naive Bayes**: se tratarán los tres algoritmos disponibles: *Gaussian, Multinomial and Bernoulli Naive Bayes*

- **Árboles de decisión**: se tratará el clasificador de Árboles de Decisión que proporciona Scikit-Learn
- **Metodos Ensemble**: se considerarán para su estudio los algoritmos *GradientBoostingClassifier* y *RandomForestClassifier*

A continuación se trata uno a uno, dando una breve explicación y probando su funcionamiento en la predicción del equipo ganador. La forma de entrenar los modelos fue explicada en 4.2.3 (Herramientas). Las pruebas se hacen manteniendo la normalización con *MaxAbsScaler* y sin transformar datos ni hacer selección de características.

La mayor parte de la teoría explicada sobre los algoritmos en esta sección se ha extraído de la propia documentación de Scikit-Learn. Para cada apartado se citará la fuente correspondiente.

4.5.1. SVM

Una máquina de vectores soporte construye un hiper-plano o un set de hiper-planos en un espacio dimensional grande o infinito. El objetivo es encontrar el hyper-plano que mayor distancia tenga a los puntos de los datos de entrenamiento, ya que cuanto mayor margen menor probabilidad de clasificar incorrectamente habrá.

Las ventajas que proporcionan las SVM [32] son las siguientes:

- Eficaz en espacios de grandes dimensiones
- Eficaz incluso cuando el número de dimensiones es mayor que el número de muestras
- Usa un subset de puntos de entrenamiento en la función de decisión (llamados vectores soporte), por lo que también es eficiente en memoria.
- Versátil: se pueden especificar diferentes funciones Kernel para la función de decisión

La principal desventaja es que SVM no proporciona estimaciones de probabilidad, deben ser calculadas usando validación cruzada de 5 iteraciones.

SVM se vuelve muy efectivo al usarse en conjunto con funciones kernels. Estas funciones permiten hacer clasificación no lineal, que provocan que el hiper-plano de mayor margen se adapte en un espacio transformado de características. Scikit Learn permite seleccionar como kernel una función lineal (el básico), una función polinómica, una función de base radial (rbf) o una función sigmoide.

Los parámetros a tener en cuenta en un modelo de SVM son:

- *C*: cuanto más ruido tengan las observaciones, menor debería ser el parámetro C ya que se correspondería con hacer una mayor regularización.
- Parámetros Kernel
  - *Degree*: grado del polinomio para función kernel polinómica
  - *Gamma*: coeficiente usado para las funciones rbf, polinómica y sigmoide.

Se procede a hacer pruebas usando el algoritmo SVC(Support Vector Classification). No obstante, este algoritmo no funciona correctamente para nuestro dataset ya que la complejidad es mayor que cuadrática con el número de muestras, por lo que al contar mínimo con 45000 datos de partidas por liga (excepto Challenger), el algoritmo se ralentiza demasiado, especialmente para valores grandes de *C*.

Por ello se implementa el algoritmo *LinearSVC*, que es similar a SVC pero sin posibilidad de elegir kernel (ya que solo es lineal) e implementado de forma que tenga más flexibilidad y pueda escalar mejor a mayor número de muestras. En la tabla 4.5 se muestran las precisiones obtenidas para distintos valores del parámetro de regularización. El entrenamiento del algoritmo es más lento cuanto mayor es el parámetro C, pero las prediciones de nuevas muestras son muy rápidas.

Como se puede observar en la tabla 4.5, los mejores resultados se dan cuando el parámetro de regularización C es 1. Si comparamos con los resultados obtenidos con Regresión Logística (tabla 4.2, en azul), se obtienen en general resultados ligeramente peores.

TABLA 4.5: PRECISIONES DE VALIDACIÓN CON SVM PARA DISTINTOS VALORES DE C

Liga	C = 0.001	C = 0.01	C = 0.1	C = 1	C = 2	C = 10
Bronce	0.5316	0.5392	0.5422	**0.5434**	0.5405	0.5008
Plata	0.5306	**0.5356**	0.5343	0.5345	0.5342	0.5006
Oro	0.5278	0.5314	0.5335	**0.5342**	0.5338	0.5008
Platino	0.5238	0.5278	0.5305	0.5316	**0.5324**	0.4996
Diamante	0.5257	0.5253	0.5262	**0.5277**	0.5254	0.5155
Challenger	0.5234	**0.5399**	**0.5399**	0.5371	0.5317	0.5096

4.5.2. k-Nearest Neighbors

Los métodos de vecinos próximos [30] se basan en buscar un número predefinido de muestras de entrenamiento lo más cercanas a un nuevo punto, y predecir la clase a partir de ellas. En *k*-Nearest Neighbors, para cada muestra de entrada $\mathbf{x}$, halla los k vecinos más cercanos y le asigna a $\mathbf{x}$ la clase mayoritaria del subset de vecinos. Para saber cuales son los vecinos más cercanos se calcula la distancia, que puede ser

cualquier medida métrica. La distancia Euclídea estándar suele ser la opción más común.

Este tipo de clasificadores son llamados métodos de aprendizaje máquina no generalizadores, ya que simplemente recuerdan todos sus datos de entrenamiento (transformados en una estructura rápida). Por ejemplo, en caso del clasificador 1-NN, este asigna a cualquier entrada $\mathbf{x}$ la categoría del vecino más cercano del set de entrenamiento:

$$d = f(\mathbf{x}) = y^{(n)}, \text{ donde} \tag{4.16}$$

$$n = \arg\min_{k} \|\mathbf{x} - \mathbf{x}^{(k)}\| \tag{4.17}$$

En caso de empates, se escoge la clase de uno de ellos de forma arbitraria. No obstante, para evitar empates se suele elegir k como un número impar.

La selección de k para cada problema es la parte más importante para este clasificador. El valor óptimo para k depende mucho de los datos: en general, un valor grande de k reduce los efectos de ruido, pero produce que los límites de clasificación sean menos claros.

Se intenta usar kNN pero ocurre lo mismo que con SVC: el algoritmo no consigue si quiera terminar de entrenarse ya que su complejidad también escala cuadráticamente con el número de pruebas. Sin embargo, en este caso no se cuenta con un algoritmo de kNN que podamos usar con una complejidad menor (como en el caso de *LinearSVC*. Por eso, se prueba a hacer selección de características con *SelectKBest*. Solo consigue entrenarse con un número bajo de características, pero aun así el algoritmo funciona extremadamente lento a la hora de predecir nuevas muestras. Por ello, se descarta el uso de kNN al necesitar un modelo que pueda predecir muestras rápidamente para usarlo en el sistema de recomendación, que es en tiempo real.

4.5.3. Naive Bayes

Los métodos de Naive Bayes [33] se basan en aplicar el teorema de Bayes con una "ingenua" (naive) suposición de independencia entre cada par de características. Se trataría de resolver la siguiente regla de decisión:

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i|y) \tag{4.18}$$

Podemos usar Maximum A Posteriori (4.1) para estimar $P(y)$ y $P(x_i|y)$, donde $P(y)$ sería simplemente la frecuencia relativa de la clase y en el set de entrenamiento. Los distintos tipos de clasificadores de Naive Bayes difieren principalmente en la

suposición que hacen respecto a la distribución de $P(x_i|y)$ (en Scikit-Learn se cuenta con 3: Gausiana, Multinomial y Bernoulli.

Los clasificadores de este tipo funcionan bien en muchas aplicaciones, y requieren pocas muestras de entrenamiento para estimar los parámetros necesarios. Además, funcionan extremadamente rápido en comparación con otros métodos de clasificación más sofisticados.

Se prueban los distintos algoritmos. El algoritmo con distribución Multinomial no puede ser usado con los datos disponibles porque se requiere que los valores de entrenamiento no sean negativos. Los resultados de los otros dos algoritmos se muestran la tabla 4.6. Para el algoritmo *BernoulliNB* se probaron múltiples valores del parámetro *alpha* (que es el parámetro de suavizado de Laplace) y en la tabla se incluyó la mayor precisión que se obtuvo de todos ellos.

TABLA 4.6: PRECISIONES DE VALIDACIÓN - NAIVE BAYES

Liga	GaussianNB	BernoulliNB
Bronce	0.5237	0.5168
Plata	0.5253	0.5106
Oro	0.5170	0.5065
Platino	0.5222	0.5079
Diamante	0.5260	0.5163
Challenger	0.5399	0.4876

Aunque la distribución Gausiana da mejores resultados que la de Bernoulli, comparado con otros algoritmos como Regresión Logística o SVM, Naive Bayes no obtiene buenos resultados en la predicción del equipo ganador de la partida. Cabe destacar que su rapidez de entrenamiento es muy alta.

4.5.4. Decision Trees

Los árboles de decisión [34] son un método de aprendizaje supervisado no paramétrico (es decir, no se basa en la obtención de un vector de parámetros como en Regresión Logística). Su objetivo es crear un modelo que aprenda reglas simples de decisión deducidas de las características de los datos para hacer predicciones.

Básicamente, se crea una estructura (llamada árbol de decisión) parecida a un diagrama de flujo, donde cada nodo interno es una prueba en un atributo, cada rama representa un resultado de dicha prueba y cada hoja (es decir, los elementos finales) son las distintas clases que existen. Aunque existe una gran variedad de algoritmos de árboles de decisión (ID3, C4.5, C5...), Scikit-Learn implementa el algoritmo CART (Classification and Regression Trees).

Las principales ventajas de los árboles de decisión [34] son:

- Fácil de entender e interpretar.
- Requiere poca preparación de los datos (otras técnicas requieren de normalización, eliminación de valores en blanco, variables ficticias...).
- Permite manejar tanto datos numéricos como categóricos.
- Funciona bien para grandes datasets.
- Usa un modelo de caja blanca. Es decir, las condiciones pueden ser explicadas por lógica booleana en aquellas situaciones observables de un modelo.
- Se puede validar un modelo usando pruebas estadísticas

Sin embargo, también presentan ciertas desventajas:

- Pueden crear árboles complejos que no generalicen los datos bien (overfitting).
- Pequeñas variaciones en los datos pueden generar árboles completamente diferentes.
- Los árboles de decisión se basan en algoritmos heurísticos, donde se hacen decisiones óptimas locales en cada nodo. Estos algoritmos no pueden garantizar que el árbol global resultante sea óptimo
- Hay ciertos conceptos que resultan difícil de aprender para los árboles de decisión como pueden ser XOR o paridad.

Se prueba el algoritmo *DecisionTreeClassifier* en el problema. Como se observa en la columna "Predeterminado" de la tabla 4.7, cuando se implementa con los parámetros por defecto, este algoritmo tiene precisión de entrenamiento del 100 % y sin embargo la precisión de validación apenas supera el 51 %. Esto quiere decir que está habiendo un claro caso de sobreajuste (overfitting), es decir, que el algoritmo se está ajustando demasiado a los datos de entrenamiento y la respuesta a muestras nuevas es muy mala. Por ello, en todos los algoritmos relacionados con árboles de decisión se incluirán datos de entrenamiento.

Para reducir el sobreajuste hay varios parámetros que se pueden modificar, como son *max_depth*, *min_samples_split*, *max_features* y *min_samples_leaf*. Se procede a dar distintos valores a todos ellos, y con el único con el que se consigue evitar el sobreajuste es con valores pequeños de *max_depth*, que es el parámetro que indica como de profundo puede ser un árbol. Cuanto más profundo, más ramas tiene y más información es capaz de capturar, y por tanto más sobreajuste se crea. En la figura 4.6 se muestra como varían las precisiones para distintos valores de este

TABLA 4.7: PRECISIÓN CON DECISION TREE

	Predeterminado		Max Depth		
Liga	Train	Valid	Depth	Train	Valid
Bronce	1.0	0.5101	9	0.6235	0.5205
Plata	1.0	0.5110	11	0.6416	0.5168
Oro	1.0	0.5037	6	0.5489	0.5184
Platino	1.0	0.5040	8	0.5662	0.5158
Diamante	1.0	0.5015	5	0.5448	0.5214
Challenger	1.0	0.4848	9	0.9769	0.5262

parámetro en la liga de Bronce. A pesar de aumentar los resultados cambiando este parámetro como se puede observar en la columna "Max Depth" de la tabla 4.7 (donde se muestra la precisión máxima obtenida y el valor de profundidad en el que se obtuvo), la precisión de validación sigue siendo bastante inferior a la obtenida con otros algoritmos.

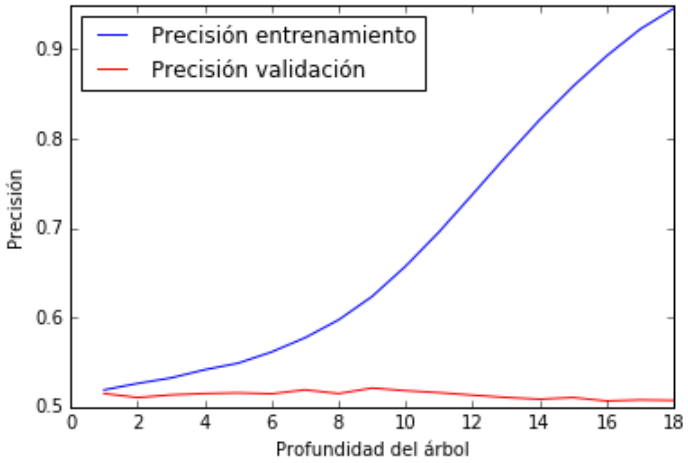


Figura 4.6: Precisión con Decision Tree en función de la profundidad del árbol

4.5.5. Ensemble Methods

Los métodos combinados [35] se basan en combinar las predicciones de varios estimadores construidos con cierto algoritmo de aprendizaje para mejorar la robustez a través de un estimador único. Se pueden distinguir en dos familias:

1. **Averaging methods**: se construyen múltiples estimadores independientes y se promedian sus predicciones. El resultado suele ser mejor que un estimador único ya que la varianza se ve reducida. El algoritmo que veremos de esta familia es *Random Forests*.

2. **Boosting methods**: los estimadores base se construyen secuencialmente y cada uno trata de de reducir el sesgo del estimador combinado. Se estudiará *Gradient Tree Boosting*

Ambos algoritmos a estudiar de métodos combinados usan como algoritmo base árboles de decisión. El parámetro fundamental a seleccionar es el número de estimadores (es decir, el número de clasificadores de árboles de decisión que se construirán de forma combinada).

Random Forests

En este algoritmo, cada árbol del conjunto se construye con una muestra recogida del set de entrenamiento [35]. Además, cuando un nodo se divide durante la construcción del árbol, la división que se eligen no es la mejor posible entre todas las características. En lugar de ello, cada división que se escoge es la mejor entre un subconjunto aleatorio de características. Como resultado de esa aleatoriedad, el sesgo del bosque aumenta levemente. Sin embargo, dado que se hace el promedio de todos los estimadores, su varianza decrece, generalmente compensando el aumento del sesgo y resultando en un mejor modelo.

Se procede a probar el algoritmo *RandomForestClassifier* con distintos valores del número de estimadores y salen resultados similares a los de Decision Tree sin especificar la profundidad máxima, es decir, que está produciéndose overfitting. Para solucionarlo, se prueban distintos valores de profundidad (esta vez funcionan mejor valores menores a los de la tabla 4.7), y se prueban de nuevo varios valores para el número de estimadores. En la tabla 4.8 se muestran los resultados.

TABLA 4.8: PRECISIONES MÁXIMAS DE RANDOM FOREST

Liga	Nº estim.	Depth	Train	Valid
Bronce	41	4	0.5714	0.5337
Plata	34	3	0.5574	0.5349
Oro	33	4	0.5615	0.5301
Platino	37	4	0.5560	0.5303
Diamante	27	4	0.5637	0.5326
Challenger	25	2	0.6853	0.5510

Se necesitan entre 20 y 40 estimadores para alcanzar precisiones de validación altas. Aunque estas se aproximan a las de Regresión Logística, no llegan a superarlas (a excepción de Diamante y Challenger). Usar valores de profundidad máxima en torno a 3-4 permite que no haya mucho sobreajuste.

Gradient Tree Boosting

Esta algoritmo construye un modelo de forma escalonada con los distintos estimadores "débiles" (generalmente Árboles de decision) y los generaliza en uno robusto, lo que permite optimizar una función de pérdida diferenciable arbitárea (en Scikit-Learn hay dos: *deviance* and *exponential*) [35]. Sus principales ventajas son que puede manejar datos de distinto tipo, que tiene un gran poder de predicción y que es robusto frente a outliers. El principal inconveniente es de escalabilidad, ya que debido a su forma de construcción es difícil de modelar de forma paralela.

Se utiliza el algoritmo *GradientBoostingClassifier*, que dispone de dos función de pérdida: *deviance* y *exponential*. El algoritmo fija por defecto la máxima profundidad de los árboles en 3, y en la mayor parte de los casos este es el valor que más precisión da, por lo que se decide dejar ese valor. El número de estimadores por defecto es 100 y la mayor precisión se obtiene alrededor de esa cifra, por lo que se probarán valores cercanos a ella.

TABLA 4.9: PRECISIÓN DE GRADIENT BOOSTING

	deviance			exponential		
Liga	Nº estim.	Train	Valid	Nº estim.	Train	Valid
Bronce	110	0.6145	0.5454	80	0.6030	0.5472
Plata	110	0.5870	0.5434	80	0.5882	0.5450
Oro	110	0.5967	0.5382	90	0.5924	0.5404
Platino	60	0.5672	0.5325	150	0.5883	0.5331
Diamante	50	0.5887	0.5315	60	0.5924	0.5323
Challenger	90	0.9374	0.5317	40	0.8731	0.5289

Como se puede observar en la tabla 4.9 La función de pérdida *exponential* da en general mejores resultados. Gradient Boosting consigue mejorar ligeramente los resultados de regresión logística y SVM. Además, aunque el entrenamiento sea un poco lento, la predicción de nuevas muestras (que es en el fondo lo que interesa para el sistema de recomendación) es rápida.

4.5.6. Selección del modelo

Una vez estudiados todos los algoritmos escogidos, es hora de seleccionar cuál se va a utilizar como clasificador. En la tabla 4.10 se muestran las mayores precisiones de validación obtenidas con cada uno de los algoritmos. Claramente, el algoritmo que más precisión proporciona en la mayoría de las ligas es Gradient Boosting. Por ello, se decide seleccionar como algoritmo de clasificación *GradientBoostingClassifier* con función de pérdida *exponential* y distinto número de estimadores para cada liga (mostrados en la tabla 4.9).

TABLA 4.10: COMPARACIÓN DE LAS PRECISIONES DE VALIDACIÓN MÁXIMAS POR MODELO

Liga	Regr. Log.	SVM	N. Bayes	Dec. Tree	R. Forest	Grad. Boost
Bronce	0.5404	0.5434	0.5237	0.5205	0.5337	**0.5472**
Plata	0.5351	0.5345	0.5253	0.5168	0.5349	**0.5450**
Oro	0.5337	0.5342	0.5170	0.5184	0.5301	**0.5404**
Platino	0.5326	0.5316	0.5222	0.5158	0.5303	**0.5331**
Diamante	0.5281	0.5277	0.5260	0.5214	**0.5326**	0.5323
Challenger	0.5234	0.5371	0.5399	0.5262	**0.5510**	0.5289

4.6. Resultados

4.6.1. Resultados test

Se ha diseñado el modelo completo del clasificador con Machine Learning Clásico. En la figura 4.7 se representa el proceso completo que se ha llevado a cabo a lo largo de este capítulo. Finalmente, la transformación de los datos no se empleará porque la precisión se reduciría. De la misma forma, de momento no se hará selección de características. Sin embargo, es importante tener en cuenta que si se necesita aumentar la velocidad del predictor a la hora de predecir una gran cantidad de muestras nuevas, seleccionando las características más importantes se podrá conseguir sin reducir demasiado la precisión del clasificador.

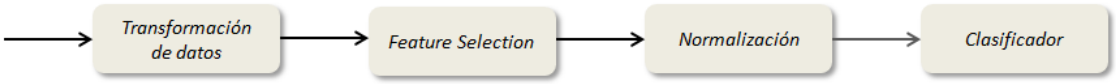


Figura 4.7: Diagrama de bloques del clasificador con Machine Learning Clásico

Dado que para las decisiones en el diseño del modelo se ha basado en el análisis de las precisiones de validación (para seleccionar de las distintas mejoras y algoritmos), es hora de verificar si el modelo diseñado responde bien a muestras que aún no se hayan probado. Por esa razón se guardó desde el principio el dataset de "test", que corresponde a un 20 % de los datos totales con los que se cuentan. Los resultados de aplicar el predictor diseñado con los datos de test se muestran en la tabla 4.11.

La precisión de test obtenida es similar a la de validación y, como se esperaba, el modelo no proporciona una precisión alta. El objetivo era intentar obtener la mayor precisión posible, aun sabiendo que el juego está programado de tal forma que las partidas sean muy difícil de predecir. Existen muchos factores, sobre todo humanos, que hacen que el resultado de una partida de un juego RTS sea muy impredecible.

TABLA 4.11: PRECISIÓN TEST DEL PREDICTOR DE MACHINE LEARNING CLÁSICO

Liga	Precisión Test
Bronce	0.5453
Plata	0.5422
Oro	0.5350
Platino	0.5275
Diamante	0.5298
Challenger	0.5289

4.6.2. Validación de los resultados

Se tiene que verificar que los resultados obtenidos en la tabla 4.11 son robustos. Para ello, una forma muy común de hacerlo es utilizar validación cruzada. El proceso que se lleva a cabo es el siguiente:

1. Se juntan los datos de entrenamiento y validación en único data set (80 % de los datos disponibles)
2. Se divide el data set en K subsets, que en este caso se decide que sean 10.
3. Se entrena el modelo con 9 de los subsets y se valida con el restante, y se hace lo mismo 10 veces de manera que todos los subsets se utilicen como validación.
4. Se calculan estadísticas (media y desviación estándar) a partir de las 10 precisiones de validación obtenidas.

En la tabla 4.12 se muestran los datos de la validación del modelo. La liga de Challenger no responde bien al modelo, probablemente porque se cuenta con muy poca cantidad de datos en comparación con el resto.

TABLA 4.12: VALIDACIÓN DEL MODELO FINAL CON VALIDACIÓN CRUZADA

Liga	Media de las precisiones	Desviación estándar
Bronce	0.5428	0.0109
Plata	0.5340	0.0066
Oro	0.5313	0.0071
Platino	0.5281	0.0053
Diamante	0.5275	0.0064
Challenger	0.4866	0.0449

Dado que la diferencia entre la media y 0.5 (el azar), es entorno a 4 veces la desviación estándar, podemos concluir que la capacidad de predicción con respecto al azar es pequeña pero estadísticamente significativa. En este tipo de juegos, esta diferencia con respecto al azar, aunque sea pequeña, puede dar una ventaja muy significativa (de la que se intentará sacar provecho con el sistema de recomendación).

5. PREDICCIÓN CON DEEP LEARNING

5.1. Introducción y objetivos

Deep Learning es una rama de Inteligencia Artificial y Machine Learning, que a través de ciertas estructuras es capaz de conformar abstracciones de datos a alto nivel. Esta rama está inspirada en la organización de sistema nervioso, y particularmente en la actividad de las capas de neuronas del neocortex [36]. La imitación de estas estructuras permiten reconocer patrones en distintos tipos de datos, desde sonidos a imágenes.

Las estructuras principales dentro de Deep Learning se llaman "redes neuronales", y hay muchos tipos de ellas. Por la naturaleza de los datos que se tratan en este proyecto y el tipo de problema, la red neuronal que más se ajusta y que se desarrollará en este capítulo es el Perceptrón Multicapa.

Se desea estudiar este tipo de estructura con el fin de mejorar los resultados de la predicción del equipo ganador de una partida de LoL obtenidos con Machine Learning Clásico. Los objetivos para este capítulo son los siguientes:

Los objetivos para este capítulo son los siguientes:

1. Entender el funcionamiento de un Percetrón Multicapa y describir la herramienta con la que poder crearlo
2. Diseñar la estructura principal de la red neuronal para nuestro problema
3. Estudiar los distintos parámetros y optimizadores disponibles para lograr obtener la mayor precisión posible

5.2. Teoría y herramientas de redes neuronales

El problema sigue siendo el mismo: clasificar una partida para decidir si gana un equipo u otro. En este caso, aunque la base matemática que hay detrás es muy parecida, se sigue otra estrategia que nos servirá para resolver el mismo problema. La estructura que utilizaremos, por adaptarse mejor a nuestro problema, es un tipo de red neuronal llamado Perceptrón multicapa. A continuación se proporciona una breve explicación sobre este modelo de clasificación y la herramienta se utilizará para construirlo.

5.2.1. Perceptrón multicapa

Un Perceptrón simple, también llamado neurona, es un algoritmo de clasificación lineal. De hecho, el algoritmo de un perceptrón es prácticamente idéntico a la regresión logística que se vio en detalle en la sección 4.2.2. Como se observa en la figura 5.1, se hace el producto escalar entre el vector de entrada y un vector de pesos, y el resultado se introduce a la función de activación que devuelve el resultado de la predicción.

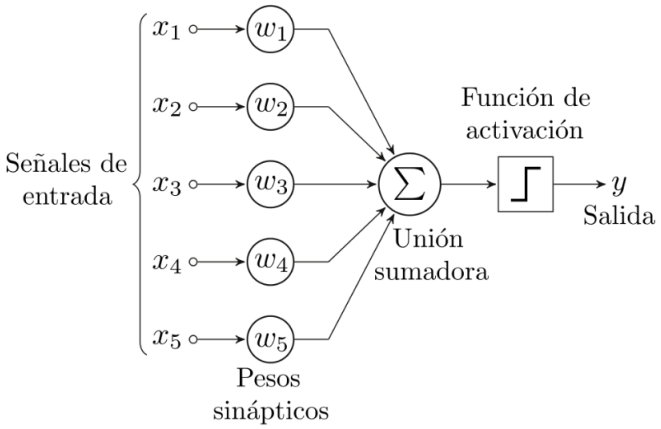


Figura 5.1: Diagrama de un perceptrón con cinco entradas. Obtenida de [37]

Un Perceptrón multicapa [38] es la combinación de varios Perceptrones simples (neuronas), agrupados en capas de diferentes niveles. Gracias a esta estructura, se puede aproximar cualquier función no lineal. Se pueden distinguir tres tipos de capas diferentes: la capa de entrada, las capas ocultas y la capa de salida.

Las neuronas de la capa de entrada se encargan de recibir las señales y propagarlas al resto de neuronas de la siguiente capa (es decir, no actúan como neuronas como tal). Las neuronas de las capas ocultas procesan las señales recibidas y transmiten el resultado a las siguientes neuronas. La última capa es la salida del Perceptrón, que devuelve el resultado de toda la red. En cada capa puede haber el número deseado de neuronas y las neuronas de una capa se conectan con las neuronas de la siguiente capa (siempre hacia delante).

El número de neuronas en las capas de entrada y salida viene definido por el problema. Sin embargo, el número de capas ocultas y el número de neuronas de cada una de esas capas deben ser elegidos. No existe forma alguna de encontrar el número óptimo para cada problema, y lo más común es seleccionar estos parámetros usando validación cruzada. Otra forma de hacerlo es con métodos de ensayo y error, que es como hará en este proyecto.

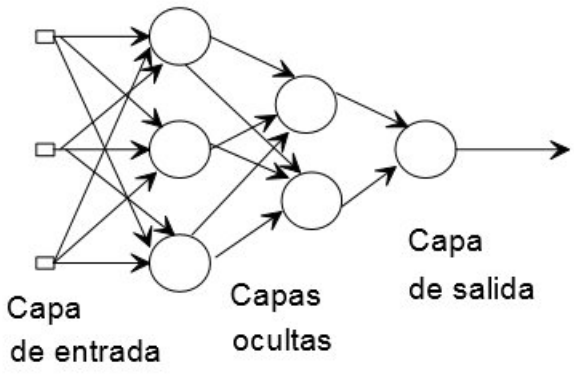


Figura 5.2: Estructura de un Perceptrón multicapa. Obtenida de [39]

La función de activación es la que permite transmitir la información entre unas neuronas y otras. Como se explicó, esta función se le aplica a la suma de los productos de las entradas que recibe. Hay varios tipos de funciones de activación, pero generalmente la elección de ellas no suele influir demasiado en la capacidad de la red para hacer una correcta predicción.

El objetivo del Perceptrón Multicapa es, gracias a los datos de entrenamiento, ajustar los pesos de cada Perceptrón simple de manera que la salida de la red sea lo más parecida a la salida deseada. Entonces, el aprendizaje de la red se formula como un problema de minimización:

$$\min_{W} E \tag{5.1}$$

Siendo W el conjunto de pesos y E una función error (también llamada función de pérdida, loss function) que evalúa la diferencia entre salidas de la red y las salidas deseadas. Existen varios algoritmos que resuelven este problema de minimización (llamados algoritmos de retropropagación) y que permiten la obtener los valores ajustados de los pesos. Si desea ampliar información sobre este tipo de algoritmos, se recomienda la lectura de [38].

5.2.2. Herramienta de redes neuronales - Keras

Para el diseño del Perceptrón Multicapa se usará *Keras*[2], una de las librerías más destacadas de Deep Learning en Python. Keras funciona por encima de otras librerías importantes de Deep Learning como *TensorFlow*[3], *CNTK*[4] o *Theano*[5]. Esto quiere decir que los modelos de Keras funcionan a alto nivel, utilizando el resto de librerías

[2] `https://keras.io/`
[3] `https://www.tensorflow.org`
[4] `https://docs.microsoft.com/en-us/cognitive-toolkit/`
[5] `http://deeplearning.net/software/theano/`

mencionadas como backend (es decir, que usa las otras librerías para operaciones de bajo nivel como productos tensoriales, convoluciones y demás). En este proyecto se usará siempre como librería backend TensorFlow, ya que es la que viene predefinida.

En Keras hay dos modelos con los que construir la red neuronal: *Sequential* y *Functional API*. El primero permite crear modelos de manera sencilla, añadiendo capa por capa, pero no permite crear modelos en los que se puedan unir una capa con cualquier otra, o modelos con múltiples inputs y outputs. Con el segundo, se permite una mayor flexibilidad pudiendo hacer lo que otro no permite, pero es ligeramente más complejo. En ambos modelos el procedimiento es el mismo:

1. Se construye el modelo de la red neuronal por capas. En cada capa se especifica el número de neuronas. Hay diferentes tipos de capas, de las cuales las que se usarán son:

   - **Input**: utilizadas para la capa de entradas. Hay que definir como parámetro la dimensión del input que se espera. En *Sequential*, esta capa se genera automáticamente (tan solo se tiene que especificar la dimension de los datos de entrada en la primera capa)
   - **Dense**: capa en la que todas las neuronas están conectadas con el resto de neuronas de la siguiente capa. Es la capa más común de una red neuronal.
   - **Dropout**: aplica *dropout* a los inputs de las neuronas. *Dropout* consiste en desactivar aleatoriamente cierto porcentaje de inputs en cada actualización durante la etapa de entrenamiento. Esto ayuda a prevenir el sobreajuste. El porcentaje viene definido por el parámetro "rate", que puede ser de 0 a 1.

   Además, en la capa se tiene que especificar la función de activación (explicada en la parte de teoría de Perceptrones múltiples). Se tratarán fundamentalmente dos funciones de activación:

   - **relu**: es la que suele dar mejor resultados. Se utiliza para las capas ocultas.
   - **sigmoid**: muy similar a la función logística (eq. 4.3). Se utiliza normalmente para la capa de salida.

2. Se compila el modelo (*compile()*). Hay que indicar los siguientes parámetros en la compilación:

   - **optimizer**: es el algoritmo que se usa para obtener los valores de los pesos en las neuronas (el algoritmo de retropropagación). Existen varios tipos que serán probados.
   - **loss**: es la función error que, como comentamos, evalúa la diferencia entre salidas de la red y las salidas deseadas. Para clasificación binaria, como es nuestro caso, se debe seleccionar "binary_crossentropy"

- **metrics**: es un parámetro opcional que sirve para monitorizar la fase de entrenamiento, es decir, mostrar aquellas métricas para comprobar como se va entrenando el modelo. Se puede insertar una lista con varias métricas. La métrica que se usará es "accuracy" (precisión).

3. Se entrena el modelo (*fit()*). Para ello se introducen los datos de entrenamiento (**x**, y), y se especifican dos parámetros:
   - **epochs**: (épocas) número de veces que se introducen todos los datos de entrenamiento (el dataset completo) al modelo.
   - **batch_size**: número de muestras que se propagan a la vez a través de la red neuronal.

Se va a utilizar tanto el modelo de *Sequential* como el modelo de *Functional API* para probar las distintas configuraciones que se propondrán durante el capítulo. Para mostrar como se crea, entrena y prueba un modelo en Keras, en la figura 5.3 se proporciona el código de un ejemplo sencillo de un Perceptrón Multicapa utilizando el modelo de *Sequential*.

```
# Importamos las funciones que usaremos de Keras
from keras.models import Sequential
from keras.layers import Dense

# Se crea el modelo Sequential
model = Sequential()

# Se introduce la primera capa oculta Dense formada por 200 neuronas
# con func. de activ. 'relu'. La capa de entrada se crea por defecto
# y solo se tiene que especificar la cantidad de datos por muestra
# (180 en nuestro caso)
model.add(Dense(200, activation = 'relu', input_dim = 180))

# Se agrega la capa de salida formada por una sola neurona con func.
# de activ. 'sigmoid'
model.add(Dense(1, activation = 'sigmoid'))

# Se compila el modelo
model.compile(optimizer = 'RMSProp',
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])

# Entrenamos el modelo
history = model.fit(X, y, epochs = 150, batch_size = 128)

# Evaluamos el modelo
score_valid = model.evaluate(X_new, y_new, batch_size = 128)
```

Figura 5.3: Ejemplo de Perceptrón multicapa con Keras - Sequential

El valor de *epochs* es un muy importante. Con un valor muy pequeño *epochs* habrá underfitting, es decir, el modelo no podrá captar la estructura de los datos porque no se ha entrenado lo suficiente. Sin embargo, con un valor demasiado alto de *epochs*, el modelo se ajustará demasiado a los datos de entrenamiento y se producirá overfitting (sobreajuste).

Keras permite mostrar como varía la precisión y la función de pérdida durante el entrenamiento. En la figura 5.4 se muestra el entrenamiento de una red neuronal con un *epochs = 150*. Lo que ocurre es un caso de sobreajuste (overfitting) ya que, como se observa, la función de pérdida a partir de la época 10 es constante, mientras que la precisión sigue creciendo en todo momento. Por lo tanto, la precisión de validación (para nuevas muestras) será notablemente inferior a la de entrenamiento.

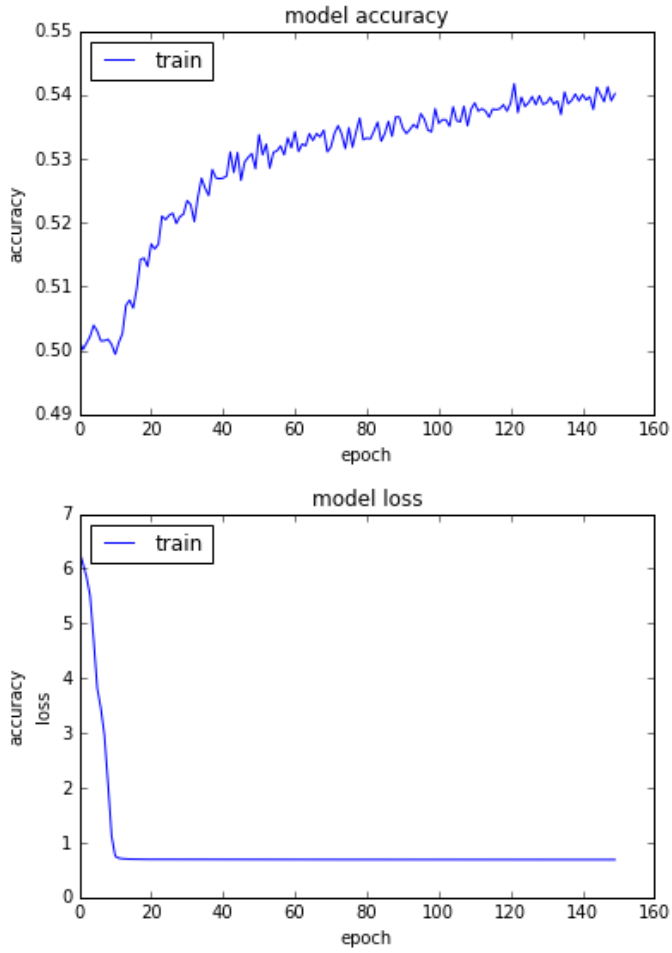


Figura 5.4: Precisión y pérdida de un modelo durante el entrenamiento

Por ello, para evitar que se produzca cualquiera de los dos problemas, Keras permite ir visualizando la precisión y el valor de loss de validación al final de cada época durante la fase de entrenamiento. En la figura 5.6 se muestra los datos que proporciona Keras durante el entrenamiento si se activa la opción de visualizar la precisión de los datos de validación. Esta información se puede almacenar de manera que pueda ser procesada posteriormente.

Además, Keras permite añadir una funcionalidad llamada "Early Stopping", que permite parar de entrenar el modelo cuando cierta condición se cumple. Es decir, en vez de entrenar el modelo el número de épocas indicado en un principio al completo, se puede indicar que cuando se produzca cierto evento (por ejemplo que la función de pérdida empiece a aumentar), el modelo se pare de entrenar inmediatamente o pasados un cierto número de épocas.

El procedimiento que se seguirá durante el resto del capítulo para hallar la precisión máxima que proporciona cada modelo es el siguiente:

- Seleccionar un valor suficientemente alto de épocas para que no haya underfitting.
- Programar la función *EarlyStopping* de manera que cuando se alcance un máximo en la precisión de validación, el modelo pare de entrenar pasados un cierto número de épocas n. El valor de n se valorará para cada caso de manera que se asegure que el modelo no puede mejorar más (es decir, que esa precisión no mejore pasados unas cuantas épocas), pero que no sea muy grande y requiera demasiado tiempo.
- Obtener la precisión máxima de validación que se obtuvo.
- Repetir varias veces este proceso y se hacer la media de todas las precisiones máximas obtenidas, ya que los valores iniciales de los pesos son aleatorios y en cada ejecución los resultados pueden ser ligeramente distintos.

En el caso de la figura 5.6, para un valor de $n = 2$, se obtendría que la precisión máxima es 0.5338 (época 7), pero para un valor de $n = 3$ se obtendría que la precisión máxima es 0.5355 (época 10). Otra forma de obtener la precisión podría ser hacer la media de las precisiones obtenidas en las últimas $n+1$ épocas. De esta forma, en aquellos casos en que las precisiones de validación oscilen en un intervalo, se podría obtener un valor para la precisión a pesar de las fluctuaciones de valores que hay entre las distintas épocas.

5.3. Modelo básico del Perceptrón Multicapa

En esta sección se describe la estructura que utilizaremos para la red neuronal. Aunque luego la modificaremos para probar distintas configuraciones(número de layers, tipo de activación en los nodos, número de salidas por nodo...), es importante partir de una estructura básica sobre la que realizar cambios. Surgen dos ideas principales para la estructura principal: una muy sencilla que intenta imitar el funcionamiento de los algoritmos de Machine Learning Clásico, y otra más compleja que intente sacar provecho de la existencia de estadísticas de campeón y posiciones de los jugadores.

```
Epoch 3/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6936 - acc: 0.5397 - val_loss: 0.6993 - val_acc: 0.5212
Epoch 4/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6885 - acc: 0.5469 - val_loss: 0.6980 - val_acc: 0.5263
Epoch 5/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6855 - acc: 0.5530 - val_loss: 0.6959 - val_acc: 0.5305
Epoch 6/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6832 - acc: 0.5590 - val_loss: 0.6960 - val_acc: 0.5307
Epoch 7/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6815 - acc: 0.5624 - val_loss: 0.6948 - val_acc: 0.5338
Epoch 8/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6799 - acc: 0.5677 - val_loss: 0.6957 - val_acc: 0.5302
Epoch 9/150
27610/27610 [==============================] - 0s 11us/step - loss: 0.6785 - acc: 0.5694 - val_loss: 0.6949 - val_acc: 0.5333
Epoch 10/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6774 - acc: 0.5746 - val_loss: 0.6944 - val_acc: 0.5355
Epoch 11/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6763 - acc: 0.5749 - val_loss: 0.6955 - val_acc: 0.5343
Epoch 12/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6752 - acc: 0.5782 - val_loss: 0.6955 - val_acc: 0.5321
Epoch 13/150
27610/27610 [==============================] - 0s 12us/step - loss: 0.6744 - acc: 0.5805 - val_loss: 0.6961 - val_acc: 0.5337
```

Figura 5.6: Información proporcionada por Keras durante el entrenamiento

5.3.1. Primer modelo básico propuesto

Se trata de una estructura muy básica con una única neurona con función de activación sigmoide. Es decir, se trata de imitar lo que hace un algoritmo de Machine Learning Clásico. De hecho, al ser la función sigmoide, el modelo sería muy similar a un algoritmo de Regresión Logística (por lo explicado en 5.2.1). Por tanto, tiene una capa de entrada (Input) con las 180 características (18 estadísticas de campeón y 10 campeones) y la capa final con una única salida. Es decir, de momento no hay capas ocultas.

Debido a su simplicidad, el modelo se construye con *Sequential*. Los parámetros con los que se construye el modelo vienen especificados en la figura 5.7. Además, los datos se normalizan con *StandardScaler* (4.4.1).

```
Batch_size = 256
optimizer = 'rmsprop'
loss = 'binary_crossentropy'
metrics = ['accuracy']
```

Figura 5.7: Parámetros para la prueba inicial de la red neuronal básica

En la tabla 5.1 se muestra la precisión obtenida para cada liga usando este primer modelo básico. Como se esperaba, los resultados son muy similares a los obtenidos con Regresión Logística en el anterior capítulo (tabla 4.10).

TABLA 5.1: PRECISIÓN DEL PRIMER MODELO BÁSICO

Liga	Train Prec.	Valid Prec.
Bronce	0.5568	0.5457
Silver	0.5450	0.5380
Gold	0.5420	0.5342
Platinum	0.5345	0.5335
Diamond	0.5430	0.5302
Challenger	0.6211	0.5321

5.3.2. Segundo modelo básico propuesto

La otra idea para el modelo básico del Perceptrón Multicapa es un poco más complicada. Las características de este segundo modelo son las siguientes:

- Está compuesta por tres capas o layers principales.
- La primera capa es la capa de entrada (*Input*), que simplemente se encarga de recibir los datos y transmitirlos a la segunda capa. Las neuronas, que no actúan propiamente como neuronas (no tienen función de activación), tienen 5 salidas (una por posición/función de cada jugador).
- La segunda capa es la capa oculta, de tipo *Dense*, que está compuesta por n nodos (neuronas), siendo n el número de estadísticas de campeón que se usen (18 en total, si no se hace selección de características). Es decir, cada neurona corresponde a una estadística y todas tienen las siguientes características:
  1. Tienen 5 inputs, uno por posición/función de cada jugador y una única salida.
  2. Su función de activación es *relu*.
- La tercera capa es la capa de salida, también *Dense*, y está compuesta por una única neurona con las siguientes características:
  1. Recibe como input las n salidas de los nodos de la segunda capa.
  2. Tiene una salida única que es la salida final del modelo.
  3. Su función de activación es *sigmoid*.

Esta configuración está pensada de manera que siempre haya una neurona por estadística de campeón, y que cada estadística tenga una entrada por cada posición del juego. De esta manera, si una estadística no tiene mucha importancia, el peso de la siguiente neurona a la que va conectada tenderá a ser bajo. De la misma forma, si cierta estadística no tiene importancia para cierta posición, esto se verá reflejado

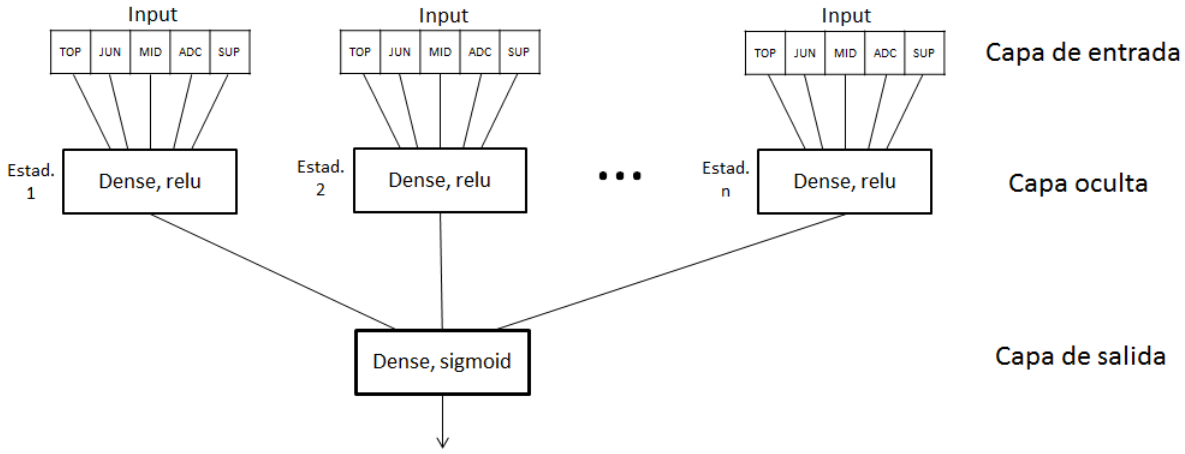


Figura 5.8: Estructura básica del Peceptrón Multicapa

en que el peso de la unión correspondiente será bajo. En la figura 5.8 se representa gráficamente la estructura.

Además, hay que tener en cuenta que solo hay 5 entradas por estadística. Esto tiene que ver con el segundo problema de la transformación de datos del capítulo anterior (4.4.2), en la que para dar a entender al modelo que existen dos equipos, se restaban o dividían las estadísticas de un equipo con otro. En este caso, lo que se hace es restar cada estadística de cada campeón con la misma estadística del campeón enemigo. El resultado de esa resta es lo que se introduce como entrada en el Perceptrón Multicapa.

Para construir este modelo se utiliza en este caso el modelo de *Functional API*, ya que se necesitan hacer uniones particulares entre las capas que con *Sequential* no sería posible. Los parámetros utilizados son los mismos que en el primer modelo (figura 5.7), salvo por el tamaño del batch que en este caso se pone de 1024 (ya que la estructura es más compleja y facilitará el aprendizaje). En la tabla 5.2 se muestran las precisiones obtenidas.

TABLA 5.2: PRECISIÓN DEL SEGUNDO MODELO BÁSICO

Liga	Train Prec.	Valid Prec.
Bronce	0.5333	0.5225
Silver	0.5338	0.5212
Gold	0.5287	0.5160
Platinum	0.5260	0.5185
Diamond	0.5350	0.5204
Challenger	0.5450	0.5203

Comparando las precisiones de ambos modelos, claramente con el primero (a pesar de su sencillez) se obtiene mayor precisiones y el aprendizaje es más rápido.

Por lo tanto, **se fija el primer modelo como el modelo principal básico**. Nuestro objetivo es, partiendo de esa arquitectura sencilla, intentar mejorar los resultados de precisión complicando la estructura y modificando los distintos parámetros del Perceptrón Multicapa.

5.4. Mejoras al Perceptrón Multicapa

Una vez se ha seleccionado el primer modelo propuesto como el modelo básico, el objetivo es intentar mejorar la precisión de este. Para ello se añadirá y configurará nuevas capas, se probarán diversos valores para los distintos parámetros y otra serie de cosas con el fin de mejorar el poder de predicción de la red neuronal.

5.4.1. Tamaño del batch

Como se ha explicado, este número fija la cantidad de muestras que entrenan la red a la vez. Es decir, que los pesos de las neuronas no se actualizan hasta que ese número de muestras se ha propagado por la red. Las siguientes conclusiones se obtienen de [40] sobre el tamaño del batch:

- Batches grandes proporcionan una estimación más precisa con algoritmos de SGD (Stochastic Gradient Descent).
- Algunas arquitecturas de redes neuronales pueden inutilizarse por valores extremadamente pequeños de batches.
- Es común dar valores en potencias de 2 para el tamaño de batch.
- El valor máximo del batch viene limitado por el hardware.

Un valor grande de batch hará que cada época dure menos, pero la red requerirá mayor cantidad de épocas para aprender. Por el otro lado, un valor de batch pequeño hará que cada época dure más pero que la red neuronal converja en menos cantidad de ellos.

Teniendo todo esto en cuenta, y tras hacer una serie de pruebas al Modelo básico, se prefiere tomar como valor $\boldsymbol{batch = 256}$ por ser un valor medio (ni bajo ni muy alto) y potencia de dos.

5.4.2. Número y tipo de capas ocultas

Número de capas ocultas y número de neuronas por capa

Uno de los motivos que hacen a las redes neuronales tan potentes con respecto a otras técnicas de Machine Learning es la capacidad de crear arquitecturas complejas

de múltiples capas. Aunque según el *Teorema de Aproximación Universal*, cualquier función continua en $\mathbb{R}^n$ se puede aproximar con una red neuronal con una única capa oculta [41], para aquellos problemas que sigan una distribución complicada un mayor número de capas podría ayudar a mejorar la precisión del modelo.

No hay ningún método para saber que número de capas es el mejor, ni qué cantidad de neuronas debe haber en cada layer, por lo que el procedimiento más usado es el de ensayo y error. Se procede a hacer una gran cantidad de pruebas, por lo que se decide hacerlas de momento solo para una liga. Se elige que sea la liga de bronce, dado que las máximas precisiones se suelen obtener para esta liga.

Partimos de las dos capas del modelo básico (la de Input con las 180 entradas y la final con una única neurona) y en primer lugar se prueba añadir una única capa oculta entre ambas, de tipo Dense, con función de activación *relu*. Se prueban múltiples valores del número de neuronas, pero ninguna consigue superar el 54 % de precisión de validación (con el básico obteníamos 54.57 %). A continuación se prueba a cambiar a la función de activación de la capa oculta a *sigmoid*, y los resultados mejoran con respecto a *relu*, llegando a aproximarse a los obtenidos con el modelo básico para números de neuronas de la capa entre 50 y 200.

Partiendo de esa última configuración, se añade otra capa oculta (entre la capa que se acaba de añadir y la capa de salida) de tipo Dense y con función de activación sigmoide. Se prueban distintos valores para el número de neuronas de ambas capas, pero se obtienen resultados similares a los obtenidos con una única capa oculta y el tiempo que tarda en converger es mayor (ya que hay un mayor número de neuronas y se necesitan ajustar más cantidad de pesos). Con función de activación *relu* en la segunda capa se obtienen peores resultados.

Finalmente se prueba a añadir una tercera capa oculta Dense, variando el número de neuronas en las tres capas ocultas. Las precisiones obtenidas son muy similares a las anteriores, pero el tiempo de entrenamiento es bastante mayor.

Se ha comprobado que añadir más cantidad de capas no aumenta la precisión de validación del modelo y hace que este requiera más tiempo para entrenarse. Se decide por tanto introducir al modelo básico una única oculta con función de activación sigmoide.

Como se aprecia en la figura 5.3, se obtienen resultados similares para distintos número de neuronas. Aunque la precisión es muy similar (y quizás peor) que el modelo sin ninguna capa oculta, se prefiere usar el modelo con la capa oculta porque proporciona mayor complejidad a la red neuronal y mayor posibilidad de experimentar con ella (por ejemplo cambiando el tipo de capa a Dropout como se hará a continuación). Además, con una única neurona como en el modelo básico se estaría haciendo prácticamente lo mismo que en el capítulo anterior, y el interés para este capítulo es explorar la capacidad de una Perceptrón Multicapa (con al menos una capa oculta).

TABLA 5.3: PRECISIONES DE VALIDACIÓN DEL MODELO BÁSICO CON UNA CAPA OCULTA CON DIFERENTE NÚMERO DE NEURONAS

Liga	150 neuronas	100 neuronas	50 neuronas
Bronce	**0.5445**	0.5423	0.5416
Silver	0.5346	0.5354	**0.5361**
Gold	0.5331	0.5336	0.**5343**
Platinum	0.5259	0.5262	**0.5296**
Diamond	0.5273	**0.5275**	0.5272
Challenger	**0.5481**	0.5436	0.5457

Uso de Dropout

Se comentó que además de Dense, existía otro tipo de capa llamado Dropout que simplemente se conecta entre una capa y otra y desconecta un porcentaje de neuronas aleatorias en cada época durante el entrenamiento y que puede ayudar al modelo a reducir el sobreajuste. Se hacen pruebas con una y dos capas ocultas. En el primer caso la capa Dropout se pone antes de la capa oculta, y en el segundo se pone entre las dos capas. Además se prueban diferentes valores para el número de neuronas por capa. Se sacan las siguientes conclusiones:

1. El modelo tarda bastante más en converger durante la fase de entrenamiento
2. Los valores del porcentaje de neuronas desconectadas que mejores resultados proporcionan están en torno al 10 %.
3. Aunque consiguen alcanzar resultados similares a los obtenidos, no consiguen mejorarlos

Por estas razones se descarta el uso de Dropout.

5.4.3. Simetría del modelo

Desde el principio, la idea era que el predictor fuera simétrico, es decir, que si se cambiase el equipo rojo por el equipo azul (es decir, los campeones del equipo azul se cambian por los del rojo y viceversa), el resultado de la partida sería el contrario. Surge la idea de intentar aumentar esta simetría para mejorar la precisión. Es importante comentar que en el anterior capítulo de Machine Learning Clásico también se implementó lo que se explicará a continuación y se obtuvieron resultados similares a los que darán a continuación, pero se ha decidido explicarlo en este capítulo para no extender demasiado el anterior.

El proceso llevado a cabo para intentar aumentar la simetría es sencillo:

- Se crean partidas manualmente en las que simplemente se invierten los equipos y los resultados de las partidas de los datos de entrenamiento.
- Se juntan ambas partidas, las originales y las creadas invirtiendo los equipos. Esto da lugar a un set de entrenamiento del doble de partidas que el original.
- Se barajan las partidas de forma aleatoria, de manera que se mezclen entre sí.

Se procede a entrenar el nuevo dataset con el modelo básico con una capa oculta añadida (con 50 neuronas). Para ello, dado que se ha duplicado el número de datos, se decide duplicar el tamaño de batch. Los resultados obtenidos son muy similares aunque ligeramente peores a los obtenidos con el dataset original (tabla 5.3). Como conclusión se puede decir que estos resultados pueden significar que los modelos están aprendiendo de manera intrínseca esta capacidad simétrica

5.4.4. Algoritmo de optimización

Este algoritmo el encargado de ayudar a minimizar la función de error (eq. 5.1) para obtener los valores óptimos de los pesos de las neuronas. Se estudiará qué algoritmos funcionan mejor para la predicción del equipo ganador de la partida. Hasta ahora hemos usado RMSprop como algoritmo de optimización, pero sin embargo existen otros algoritmos de optimización en Keras: *Adagrad*, *Adelta*, *Adam*, *Adamax* y *Nadam*.

Si el lector se interesase por el funcionamiento de cualquiera de estos algoritmos, se recomienda la lectura de [42]. Para probar los algoritmos se usará tanto el modelo básico como el modelo con una capa oculta de 50 neuronas. Los resultados obtenidos con los distintos algoritmos, tras hacer 3 pruebas por algoritmo, modelo y liga (216 pruebas en total), se encuentran en las tablas 5.4 y 5.5 (téngase que en cuenta que al haber tan pocas partidas en la liga de Challenger, los resultados son bastante aleatorios).

TABLA 5.4: PRECISIONES DE VALIDACIÓN DEL MODELO BÁSICO POR ALGORITMO DE OPTIMIZACIÓN

Liga	RMSprop	Adagrad	Adadelta	Adam	Adamax	Nadam
Bronce	0.5457	0.5428	0.5420	0.5451	0.5437	**0.5458**
Silver	0.5380	0.5365	0.5365	0.5386	0.5368	**0.5394**
Gold	0.5342	0.5326	0.5338	0.5352	0.5363	**0.5374**
Platinum	0.5335	0.5321	0.5346	**0.5348**	0.5338	0.5332
Diamond	0.5302	0.5272	0.5278	0.5306	**0.5308**	0.5299
Challenger	0.5321	0.5381	0.5234	0.5271	0.5201	**0.5383**

TABLA 5.5: PRECISIONES DE VALIDACIÓN DEL MODELO CON UNA CAPA OCULTA DE 50 NEURONAS

Liga	RMSprop	Adagrad	Adadelta	Adam	Adamax	Nadam
Bronce	0.5416	**0.5434**	0.5427	0.5417	0.5417	0.5416
Silver	0.5361	**0.5371**	0.5361	0.5365	0.5364	0.5348
Gold	0.5343	0.5348	**0.5365**	0.5341	0.5344	0.5330
Platinum	0.5296	**0.5332**	0.5316	0.5322	0.5323	0.5253
Diamond	0.5272	**0.5303**	0.5273	0.5281	0.5323	0.5297
Challenger	**0.5474**	0.5426	0.5400	0.5392	0.5401	0.5426

Se obtienen las siguientes conclusiones del estudio de los diferentes algoritmos:

1. Todos los algoritmos proporcionan en general precisiones muy similares, pero para el modelo básico el que ligeramente da mejores resultados es *Nadam*, y para el modelo con la capa oculta es *Adagrad*

2. *Adagrad* y *Adadelta* obtienen resultados parecidos, ya que el segundo es una extensión del primero. Su tiempo de entrenamiento es un poco superior que el resto de los algoritmos, pero los resultados entre diferentes ejecuciones de un mismo modelo y liga suelen ser similares entre sí, siendo bastante estable.

3. *Adam* y *Adamax* también obtienen resultados muy similares (ya que al fin y al cabo uno es una variante ligera del otro). Ambos algoritmos convergen muy rápido y las precisiones en cada ejecución son muy similares entre sí. *Nadam* por su parte, que también es un algoritmo parecido a ellos, tarda un poco más en converger y sus resultados no son tan similares a los otros dos.

Por todo ello, aunque el modelo básico pueda dar precisiones algo superiores, este funciona de manera muy similar a la regresión logística estudiada en el capítulo anterior, por lo que se decide seleccionar el modelo con la capa oculta de 50 neuronas y algoritmo de optimización *Adagrad*, para así comprobar como responde una estructura más compleja de Deep Learning a las pruebas finales con los datos de test.

5.5. Resultados

Se ha terminado de diseñar el Perceptrón Multicapa. De la misma forma que en el capítulo anterior, no se puede llegar a asegurar que la estructura obtenida de Deep Learning sea óptima para este problema de clasificación, pero es la que la que mejores resultados ha proporcionado a las pruebas y mejoras propuestas. Se trata de de una estructura muy sencilla formada por las siguientes características:

- Tiene tres capas:
  1. Capa de entrada con 180 entradas.
  2. Capa oculta *Dense* de 50 neuronas con función de activación *sigmoid*.
  3. Capa de salida con función de activación sigmoide y una neurona que corresponde a la salida del Perceptrón Multicapa.
- Algoritmo de optimización *Adagrad*
- Tamaño de batch: 256

Además, es importante comentar que los datos se normalizan con *StandardScaler* antes de introducirse a la red neuronal y que se probó entrenar la red neuronal con menos cantidad de características usando el procedimiento de Feature Selection explicado en 4.4.3, pero no mejoró la precisión.

Los resultados finales para el modelo de Deep Learning serán son los obtenidos con el set de test, que no se ha utilizado para decidir entre los distintos parámetros y diseños del Perceptrón Multicapa. Hasta ahora, para hallar las precisiones se obtenía la mayor precisión de un modelo, y se ejecutaba cada modelo al menos 3 veces para hacer una estimación de la precisión máxima media. Sin embargo, ahora necesitamos dejar el modelo entrenado para probarlo con el set de test. Para ello, se programa una función "callback" para que cuando la precisión de validación sea superior a cierto valor, se pare de entrenar el modelo. Los valores que se seleccionan para cada liga son los indicados en la columna de Adagrad de la tabla 5.5. En la tabla 5.6 se muestran los resultados de aplicar el modelo final al dataset de prueba.

TABLA 5.6: PRECISIÓN FINAL DEL PREDICTOR DE DEEP LEARNING

Liga	Precisión Test
Bronce	0.5478
Plata	0.5412
Oro	0.5311
Platino	0.5238
Diamante	0.5286
Challenger	0.4812

Si se comparan estos resultados con los obtenidos en Machine Learning Clásico (tabla 4.11), los obtenidos en este capitulo son inferiores para la mayoría de las ligas. Además, estos resultados no son tan estables como los otros (que siempre devuelven la misma precisión al usar el modelo con los datos de test), ya que en entre distintos entrenamientos del modelo los pesos de la red neuronal se inicializan aleatoriamente

y los resultados pueden variar ligeramente (aunque la estructura y parámetros del modelo sean exactamente los mismos).

Por esa la razón, aunque por analogía con el capítulo de Machine Learning Clásico se podría haber hecho la validación de este modelo final con validación cruzada para poder decir que los resultados obtenidos en la tabla 5.6 son estadísticamente significativos, la validación no va a hacer que se mejoren los resultados con respecto a los obtenidos con Scikit-Learn.

6. ALGORITMO DE RECOMENDACIÓN

6.1. Introducción y objetivos

Una vez se ha estudiado a fondo la forma de predecir el equipo ganador de una partida, que es la parte más compleja y la requiere más trabajo, en este capítulo se diseñará el sistema de recomendación que hace uso de esa capacidad de predicción.

Nuestra misión es proporcionar una recomendación sea cual sea el número de campeones que hayan sido seleccionados durante la fase de Draft, que fue explicada en detalle en la introducción del proyecto. Para el caso particular del último jugador en seleccionar (jugador 10, ver figura 6.1), es automático que el mejor campeón es aquel que al introducirlo junto al resto de los 9 campeones seleccionados al predictor obtenga la mayor probabilidad de ganar.

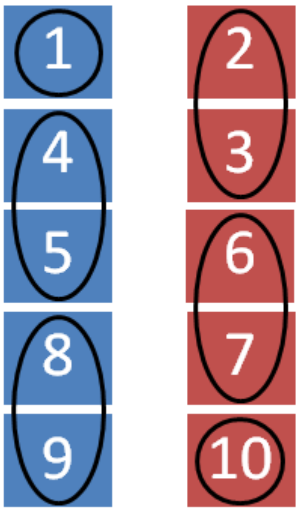


Figura 6.1: Representación del Draft. Los círculos y elipses representan los turnos de selección. Los números representan el número de cada jugador.

Sin embargo, hacer la recomendación para el resto de turnos del Draft no es algo tan simple. Por una parte, el clasificador solo es capaz de hacer una predicción dados 10 campeones y no se puede introducir una número menor de campeones para hallar probabilidades de ganar. Por otra parte, hay que tener en cuenta que los jugadores enemigos son inteligentes y también intentarán hacer la mejor elección basada en el campeón que se seleccione. Se tiene que diseñar el sistema de recomendación teniendo esto siempre en cuenta, por lo que se hará uso de teoría de juegos para

tomar siempre la mejor decisión.

En este capítulo capítulo se hará lo siguiente:

1. Dar una breve explicación sobre las técnicas de teoría de juegos que se usarán para entender el funcionamiento del algoritmo.
2. Diseñar y elaborar el algoritmo de recomendación.

6.2. Árbol de juego y método minimax

6.2.1. Árbol de juego

El Draft se trata de una fase que funciona por turnos. Un juego que funcione por turnos, como puede ser el ajedrez o las damas, puede representarse a través de una estructura de árbol, que es muy similar a un árbol de decisión. En un árbol de juego, los nodos representan decisiones a tomar, y cada una de las ramas que salen de cada nodo representa una posible decisión. Un nodo terminal (aquel del que no salen más ramás) representa una clase o resultado.

El resultado del juego puede ser cualquier cualquiera (ganar 5 puntos, ganar, empatar...). Por ejemplo, en el ajedrez habría dos posibles resultados (sin contar los que terminan en tablas): ganar o perder. En muchos casos, el árbol es tan grande que es computacionalmente imposible construirlo. En otros casos, como en el LoL, ni siquiera se sabe como va a acabar la partida al final de la ronda. Por eso surge la necesidad de puntuar el juego y asignar una puntuación a cada estado del juego.

Imaginemos un ejemplo sencillo: hay dos jugadores que se tienen que repartir entre ambos 4 elementos distintos (A, B, C y D) para empezar a jugar usando esos elementos. En cada turno uno, cada jugador debe escoger un elemento, teniendo en total 3 turnos (ya que en el último lógicamente el elemento sobrante se asigna automáticamente). Cómo no se tiene seguridad de cómo va a acabar el juego sea cual sea los elementos que escoja cada uno (cómo en el caso de los campeones en el LoL), se tiene que encontrar una función que proporcione a posible combinación i de estos elementos una puntuación p_i distinta.

La representación del árbol de este juego se puede ver en la figura 6.2. En este caso cada nodo representa la elección de un elemento de cada jugador, y los nodos terminales representan una puntuación de la combinación de todas las elecciones tomadas. Aunque no se ha completado el árbol por temas de extensión y estética, se representa perfectamente cómo para cada decisión de un jugador, el otro tiene otras posibles opciones, y así hasta llegar a seleccionarse todos los elementos y obtener el resultado del juego (que en este caso se representa por la puntuación p_i de cada combinación de elementos).

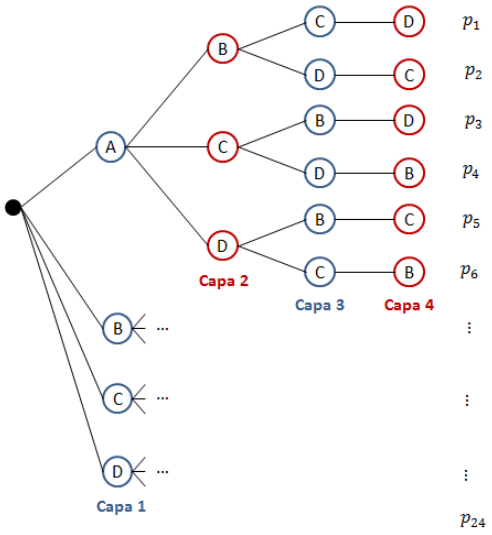


Figura 6.2: Representación del árbol de juego del ejemplo propuesto

6.2.2. Método Minimax

El método minimax [43] es una estrategia de decisión que trata de minimizar la pérdida máxima que se puede llegar a sufrir. Es decir, se trata de elegir la mejor opción para ti teniendo en cuenta que el jugador adversario elegirá aquella opción sea la mejor para él, y por lo tanto la peor para ti. Este método asume entonces que el contrincante siempre va a saber qué opción es la peor para ti.

En el ejemplo propuesto anteriormente, imaginemos que el jugador 1 es el primero en seleccionar (es decir, se representaría por el color azul en la figura 6.2) y selecciona el elemento A. Supongamos que el jugador 2 quiere obtener la mayor puntuación p_i posible. Entonces, asumiendo que el jugador 1 también conoce la puntuación de cada combinación, el jugador 2 intentará escoger entre B, C y D, sabiendo que después el jugador 1 escogerá de tal forma que el jugador 2 obtenga la menor puntuación (es decir, que le haga perder puntuación). Por ello, el jugador 2 tiene que minimizar la mayor pérdida posible o lo que se traduce en escoger el elemento que mayor puntuación le de sabiendo que el jugador 1 va a intentar minimizar su puntuación. Esto se traduce en que la mejor decisión para el jugador 2 es:

$$max\left[min(p_1, p_2),\ min(p_3, p_4),\ min(p_5, p_6)\right] \tag{6.1}$$

Por tanto, si en un juego se cuenta con una "función de pérdida" que pueda proporcionar puntuaciones a una cierta capa del árbol, se puede resolver el árbol y obtener una decisión minimizando o maximizando en cada capa del árbol de juego (según corresponda), de ahí el nombre "Minimax"

6.3. Algoritmo propuesto

6.3.1. Consideraciones iniciales

El Draft, al ser una fase por turnos, se puede representar con un árbol. La función de pérdida que hace falta para aplicar Minimax en el árbol se basa en el predictor que hemos diseñado en los dos capítulos anteriores (de ahí que sea tan fundamental para el sistema de recomendación y que se le haya dedicado tanto tiempo). Aunque las precisiones obtenidas en el predictor no sean muy altas, es importante aclarar que para el algoritmo propuesto se asume que el predictor que se vaya a usar es fiable.

Como se cuenta con 139 campeones, aunque hay 10 campeones que se banean al principio del Draft, el árbol que se formaría para todo el proceso completo de la selección de campeón sería extremadamente grande, ya que hay que tener en cuenta que en cada selección hay como mínimo 120 posibles decisiones. Por tanto, no es posible hacer una predicción para cada caso ya que requeriría demasiado tiempo computacional, y existe un tiempo máximo durante el Draft para hacer la selección del campeón. Se debe recurrir a técnicas de muestreo.

Función de muestreo: get_probs( )

En este caso, dados unos campeones que ya han sido seleccionados en un Draft, se entiende por muestreo la generación de partidas completas (generando campeones aleatorios que completen cada partida), de manera que con un número suficiente de ellas se pueda obtener una probabilidad representativa de ganar seleccionando cada campeón.

Antes de explicar la función *get_probs()*, es conveniente recordar que el clasificador se diseñó para introducir los campeones en el orden según sus posiciones (ver figura 3.3). Sin embargo, durante la fase de selección de campeones, un jugador solo puede ver a qué posición va cada uno de su equipo, pero no puede saber con seguridad a que posición va cada campeón enemigo. Normalmente no es difícil suponer a qué posición va cada campeón, pero hay ocasiones en las que como un campeón puede jugar en distintas posiciones, puede haber cierta inseguridad. Por ello, se cuenta con una lista (obtenida de [44]) que contiene las posiciones donde puede o suele ir cada campeón. Se asignará de manera automática una posición para cada campeón al que no se le haya especificado una, basándose en esa lista y en las posiciones ya fijadas en su equipo (para que no se repitan posiciones).

La función de muestreo *get_probs()* servirá para hallar la probabilidad de ganar seleccionando cada campeón. El funcionamiento es el siguiente:

1. Recibe cuatro parámetros:

- *list_selected*: Una lista de los campeones seleccionados (es decir, menos de 10 campeones), con las posiciones de cada uno. Si para uno no se conoce la posición se hará lo que se acaba de explicar.
- *list_possible*: Una lista de los campeones de los que se desea obtener la probabilidad de ganar.
- *n_muestras*: Es el número de partidas aleatorias que se generarán para cada campeón.
- *position*(Opcional): La posición para la que se desea hallar las probabilidades de ganar de cada campeón. Si no se especifica, la posición será elegida de forma aleatoria (según las posiciones ya seleccionadas).

2. Se crea un array *probs* que tiene un tamaño igual que el de *list_possible*.
3. Para cada elemento i del array *probs* se hace lo siguiente:
   - Se crea una matriz de partidas con un número de filas igual a *n_muestras*. En esta matriz todas las filas están formadas por los campeones seleccionados (en sus respectivas posiciones) y el campeón a probar (que corresponde al elemento de i de *list_possible*. Además, para las posiciones que queden libres (es decir, las que no se haya seleccionado), se generan campeones aleatorios (que no hayan sido seleccionados ni baneados).
   - La matriz de campeones se transforma en la matriz de estadísticas (es decir, 180 estadísticas de campeón por fila/partida) y se usa el predictor para obtener la probabilidad de ganar del equipo al que pertenece el jugador en cada partida.
   - Se hace la media de todas las probabilidades de cada fila, y esa media se inserta en el elemento i del array *probs*.
4. Finalmente se devuelve el array *probs* con la probabilidad de cada campeón de ganar la partida.

Turno de selección y número del jugador

Antes de proceder con el desarrollo del algoritmo, es importante diferenciar entre lo que se entiende en este algoritmo por el turno de selección y el orden del jugador. En el Draft, durante la fase de selección hay 6 turnos diferentes, que se representan por círculos y elipses en la figura 6.1 y que cada uno dura 30 segundos. En los turnos representados por elipses, los dos jugadores correspondiente seleccionan a la vez (es decir, tienen ambos el mismo tiempo para seleccionar).

Por tanto, a la hora de hacer la recomendación, se deben distinguir dos casos: que el jugador con el que se está eligiendo a la vez haya seleccionado un campeón o que no lo haya seleccionado. Esto es muy importante porque si el jugador aliado

no lo ha seleccionado, la recomendación tiene que suponer que el campeón que va a elegir va a ser óptimo. Y si lo ha seleccionado, el campeón seleccionado se cuenta como un dato más por lo que la recomendación será más precisa.

Por eso se diferencian los números de jugadores, ya que por ejemplo para el segundo turno, en la que eligen los jugadores 2 y 3, si la recomendación se tiene que hacer para el jugador 2, se debe tener en cuenta que no se ha seleccionado el campeón del jugador 3. Y sin embargo, si es al jugador 3 al que se le hace la recomendación, se asume que el jugador 2 ya ha seleccionado su campeón.

En definitiva, los jugadores 2, 4, 6 y 8 representan que los jugadores con los que seleccionan a la vez (3, 5, 7, y 9 respectivamente) no han elegido campeón aún. Y de la misma forma, los jugadores 3, 5, 7, y 9 representan que en el turno en el que están eligiendo, su compañero ya ha seleccionado el campeón.

6.3.2. Desarrollo del algoritmo

Recomendación para el jugador 10

Como se explicó, la recomendación para el jugador 10 (el último que elige, que puede ver el resto de campeones seleccionados de la partida) es bastante sencilla. Simplemente para cada campeón que no haya sido seleccionado ni baneado, se obtiene la probabilidad de ganar del equipo introduciendolo al predictor junto con el resto de los 9 campeones seleccionados (formando la partida de 10 campeones), cada uno en la posición correcta. Se obtiene aquel campeón que proporcione una mayor probabilidad de ganar la partida. Este será el campeón que se recomendará al jugador.

Suponiendo que se han baneado 10 campeones, al haber un total de 139 campeones y como se han seleccionado 9 campeones, se tendría que hacer la predicción de 120 partidas. Esto es totalmente asequible de hacer ya que el tiempo de computación necesario para hacerlo es mínimo.

Recomendación para el jugador 9

Ahora la recomendación se complica un poco. Hay que tener en cuenta que el jugador 10 selecciona después del jugador 9, y que intentará escoger el mejor campeón posible para ganar la partida. Por ello es necesario construir el árbol de juego que, como se muestra en la figura 6.3, sería relativamente sencillo.

Para la recomendación se utilizaría entonces el método Minimax. Se haría de la siguiente manera:

1. Para cada campeón i que pueda seleccionar el jugador 9, se hace lo siguiente:

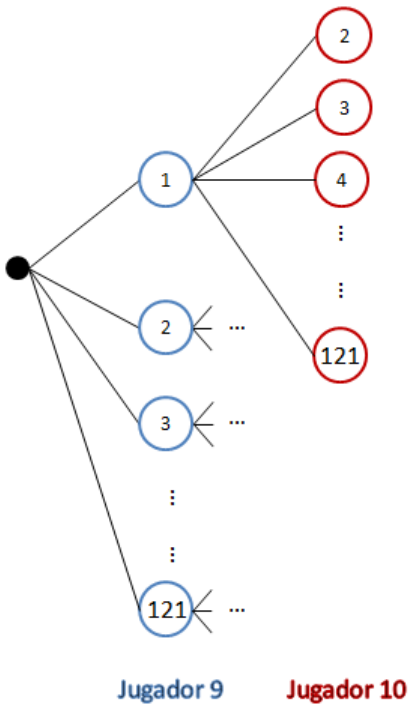


Figura 6.3: Árbol de juego para selección del jugador 9

- Se forman partidas compuestas por los 8 campeones seleccionados, el campeón i, y uno de los campeones restantes.
- Se obtiene la probabilidad para cada uno de los campeones restantes introduciendo las partidas al predictor y se selecciona la menor probabilidad.

2. Una vez obtenidas las probabilidades mínimas para cada campeón i, se selecciona el campeón que tenga la probabilidad máxima de todas ellas. Ese será el campeón a recomendar.

Este proceso requerirá, teniendo en cuenta que se banean 10 campeones al principio del Draft, una cantidad de $121 \cdot 120 = 14520$ predicciones. Aunque ahora tardará en hacerse unos pocos segundos, este proceso de recomendación sigue siendo totalmente asequible de hacer.

Recomendación del resto de jugadores

Si siguiéramos el mismo procedimiento del jugador 9 por ejemplo para el jugador 8, habría que hacer un total de $122 \cdot 121 \cdot 120 = 1771440$ predicciones. Para el jugador 7 serían 217 millones de predicciones, y así en adelante. Esto ya sería un problema ya que el tiempo de computación para la predicción de una cantidad tan grande de

partidas sería mayor que el tiempo disponible para hacer la selección del campeón en el Draft.

Es aquí donde se necesita la función *gen_probs( )* que se ha definido anteriormente. La manera más sencilla de hacer una recomendación para cualquier jugador en cualquier turno es:

1. Introducir a la función *gen_probs( )* los siguientes parámetros:
    - La lista de campeones que hayan sido seleccionados hasta ese momento con las respectivas posiciones (si se conocen).
    - La lista de posibles campeones formada por aquellos campeones que no hayan sido ni seleccionados ni baneados.
    - Un valor para número de muestras razonablemente alto (cuanto más alto más preciso, pero mayor carga computacional)
    - La posición en la que va a jugar el jugador a recomendar.
2. Obtener la lista de probabilidades para cada posible campeón y recomendar al campeón que tenga la mayor probabilidad de ganar.

Este proceso requeriría un total de *n_champs x n_muestras* predicciones, siendo *n_champs* el número de posibles campeones a seleccionar. Asumiendo que el predictor es fiable y ajustando apropiadamente el número de muestras, se puede llegar a tener una recomendación decente sin demasiada carga computacional.

Sin embargo, aunque la construcción del árbol de juego completo no es realizable para la mayor parte de los jugadores en la selección de campeón, existe una técnica llamada **podar el árbol** (inspirada de la Poda alfa-beta [45]) que puede permitir representar algunas capas del árbol. Esta técnica se basa en eliminar (podar) las ramas del árbol innecesarias para reducir el número de nodos terminales.

Como se representa en la figura 6.4, se trataría de obtener las probabilidades de cada campeón con la función *gen_probs( )* y seguir construyendo el árbol solo con aquellos campeones que obtengan las mayores o menores probabilidades (según si en la capa del árbol se tiene que maximizar o minimizar). Cuando se haya terminado de construir el árbol con un cierto número de capas, se aplica el método Minimax.

Podando el árbol podemos conseguir reducir el número de predicciones considerablemente. Por ejemplo, si se desean crear dos capas del árbol para el jugador 7, haría falta el siguiente número de predicciones N:

$$N = 123 \cdot M \cdot R \cdot M = 123 \cdot R \cdot M^2 \tag{6.2}$$

donde R es el número de ramas que no son podadas y M el número de muestras usado para la hallar la probabilidad de cada campeón.

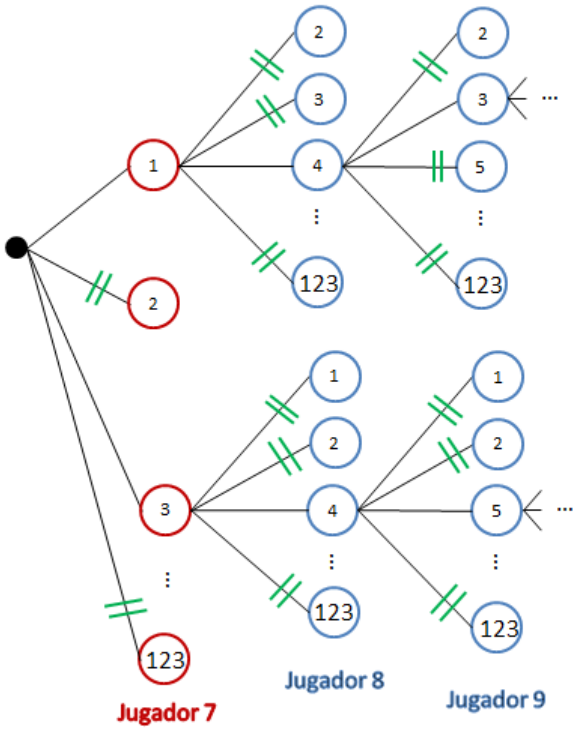


Figura 6.4: Ejemplo de árbol podado

A mayor número de muestras y mayor número de ramas sin ponderar, más rigurosa será la recomendación, pero mayor tiempo de procesado requerirá. La finalidad sería conseguir el mayor número de muestras que hagan que el tiempo total de la recomendación sea inferior al tiempo máximo que se dispone para elegir un campeón (30 segundos). Esto dependerá de la capacidad de cada ordenador. Además, lógicamente cuantos más capas del árbol se construyan, más tiempo se necesitará para hacer la recomendación.

6.3.3. Algoritmo final

Para el algoritmo final propuesto, se decide que el árbol que se construya tenga dos capas máximo para reducir todo lo posible el tiempo de procesado pero teniendo un número de muestras suficientemente grande como para que la recomendación sea fiable. La elaboración del algoritmo para un mayor número de capas es automática partiendo de este (y sería conveniente usar recursión).

En la figura 6.5 se muestra el diagrama de flujo del algoritmo completo de recomendación. Para solicitar la recomendación se tienen que introducir los siguientes parámetros:

- **list_selected**: lista de los campeones que hayan sido seleccionados junto con las posiciones de cada uno (si se conocen).
- **list_bans**: la lista de los campeones baneados antes de la selección de cam-

peones.

- **position**: la posición para la cuál se desea hacer la recomendación. El equipo al que se desea hacer la recomendación se conoce el número de campeones seleccionados (si hay 1, 2, 5, 6 o 9 campeones seleccionados sería que la recomendación es para un jugador del equipo rojo, otro número sería el equipo azul)
- **n_samples**: número de partidas aleatorias (muestras) que se generarán para cada campeón en la función *get_probs()*.
- **n_branch**: número de ramas tendrá la primera capa (es decir, número de ramas que no se podarán).

Como se explicó, hay que ajustar los parámetros *n_samples* y *n_branch* de manera que se pueda dar una recomendación fiable en el tiempo que se dispone, por lo que su valor dependerá de cada dispositivo.

Volviendo al algoritmo, si se introducen 8 o 9 campeones, quiere decir que hay que hacer la recomendación para el jugador 9 o 10, respectivamente. La recomendación para estos jugadores se explicó previamente. Si no se trata de los jugadores 9 y 10, hay que distinguir entre aquellos jugadores cuyo compañero del mismo turno ha seleccionado el campeón, y aquellos que su compañero aún no ha seleccionado.

Los jugadores que no ven el campeón de su aliado seleccionado aún son los jugadores 2, 4, 6 y 8 (ver figura 6.1), por lo que el número de campeones que han sido seleccionados para cada uno son 1, 3, 5 y 7, respectivamente. Para estos jugadores, como el siguiente campeón en seleccionar es el de sus compañeros, el árbol resultante estará formado por dos capas del mismo equipo (ver figura 6.4). Por tanto habrá que maximizar en ambas capas del árbol (*oper* = "max").

Por el otro lado, para los campeones restantes que son los que sí ven el campeón aliado seleccionado, el árbol resultante será similar al de la figura 6.3 (pero podado), y habrá que minimizar la segunda capa (*oper* = "min") y maximizar el primero.

La forma de podar el árbol es primer lugar obtener las probabilidades (*probs*) para cada campeón en la primera capa usando la función *get_probs()*. Una vez que se obtienen, se eligen los campeones que mayor probabilidades obtienen. La cantidad de campeones elegidos viene fijada por el parámetro *n_branch*. El resto de campeones no seleccionados se eliminan (se podan).

Para cada uno de los campeones elegidos se hace el mismo proceso: se mete el campeón a la lista de campeones seleccionados (*list_sel.push(champ)*) y esta lista se introduce a la función *get_probs()* para obtener las probabilidades. En este punto, si se quisieran hacer más capas, el proceso sería el mismo: podar los campeones con menores o mayores probabilidades(según el tipo de capa).

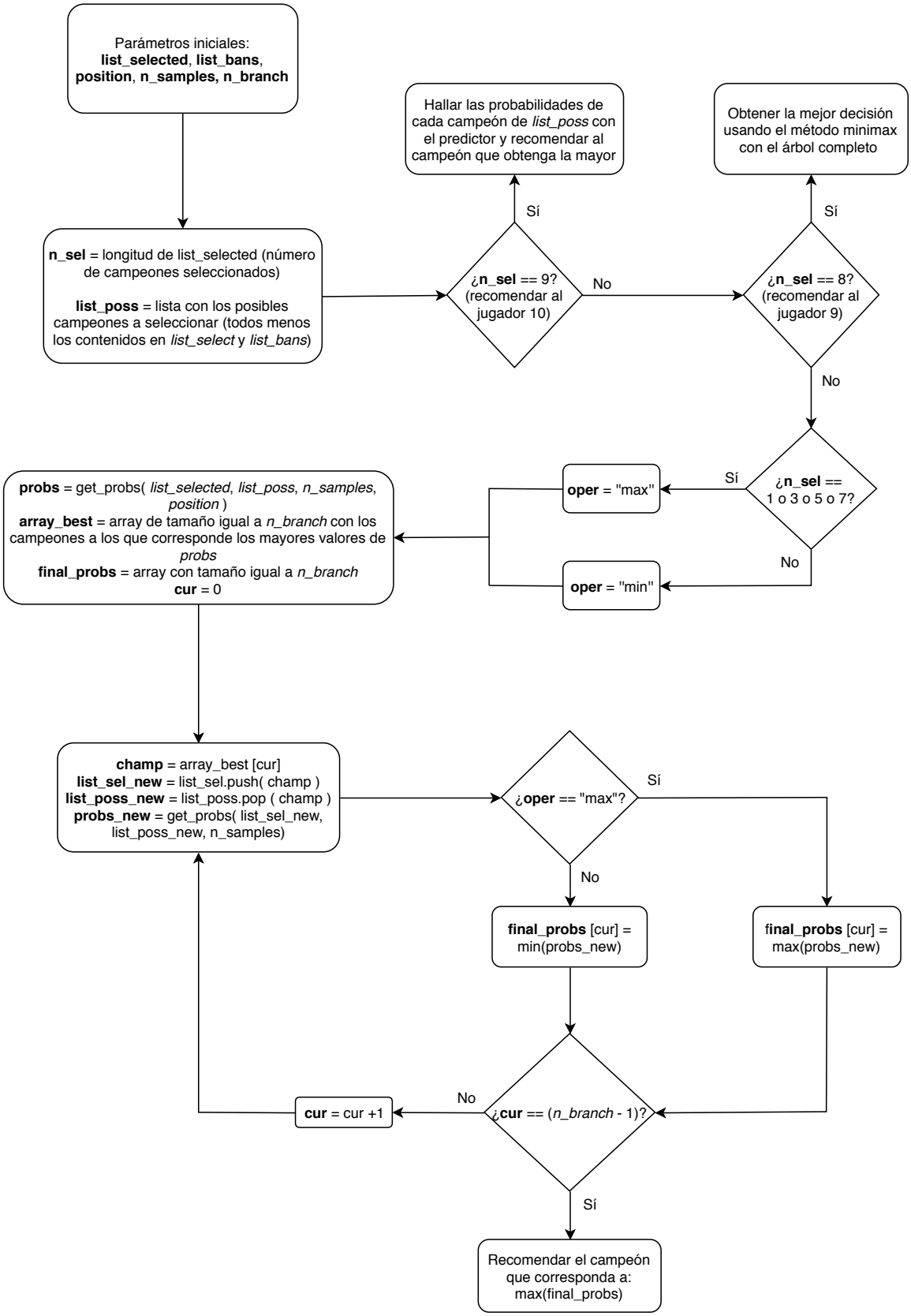


Figura 6.5: Diagrama de flujo del algoritmo de recomendación para árbol con dos capas

Sin embargo, al ser la última capa (ya que solo se desean tener 2), se aplica directamente el método Minimax. Se obtienen el máximo o mínimo (según el parámetro *oper* ya explicado) de cada una de las probabilidades de los campeones de la última capa, y finalmente se obtiene el máximo de todos ellos.

El campeón al que corresponde la probabilidad obtenida de esa maximización será finalmente el campeón que se recomendará al jugador.

6.4. Comentario sobre la validación del algoritmo

Es muy importante resaltar que este algoritmo es un prototipo, y que no ha sido validada su eficacia. La razón principal para no poder validarlo es la dificultad que implica. Una opción sería asumir que la probabilidad de victoria proporcionada por el predictor es exacta (es decir, que las precisiones son perfectas) y evaluar el sistema haciendo simulaciones. Otra opción sería hacer estudios con jugadores reales en los que se comparen las decisiones u opiniones de estos, que se basan en su experiencia dentro del juego, con las recomendaciones proporcionadas por el algoritmo.

En cualquier caso, la validación del sistema de recomendación es un problema en sí mismo y queda fuera del alcance de este proyecto, por lo que se propone como una línea futura de investigación.

7. CONCLUSIONES Y LÍNEAS FUTURAS

7.1. Conclusiones

Los objetivos del proyecto se han cumplido: se ha diseñado un sistema de recomendación para fase de selección de campeones del juego League of Legends. Su funcionamiento se basa en un predictor que proporciona la probabilidad de ganar de un equipo para una partida completa, y que se usa para resolver el árbol de juego de la fase de selección utilizando el algoritmo Minimax. Para diseñar este predictor se han seguido estrategias de Machine Learning y Deep Learning, proporcionando la primera mejores resultados. La dificultad del problema hace que la precisión del predictor final sea baja. Sin embargo, pequeñas diferencias con respecto al azar pueden proporcionar ventajas, y estas se aprovechan en el sistema de recomendación.

Entre las aportaciones que se han hecho en este proyecto destaca un método de extracción de datos a través de la API que clasifica partidas en cuanto al nivel de los jugadores, un estudio del funcionamiento de distintos algoritmos y técnicas comunes de Aprendizaje Automático para el problema de predicción del equipo ganador de una partida, y sobre todo una propuesta de un sistema de recomendación basado en técnicas de teoría de juegos que está en la línea de lo que se sugiere en [25] y que puede ayudar en las decisiones de los jugadores en tiempo real.

Es importante destacar que el sistema creado puede aplicarse a otros videojuegos con procesos de selección de personajes similares. Buena parte del software puede ser reutilizado para otros juegos, por lo que decide publicar el código del proyecto en `https://github.com/SergioEG/LoLRecommend` (donde se incluye la implementación del algoritmo de recomendación en Python) de manera que la gente pueda utilizarlo. Puesto que el sistema se basa en la utilización de una función de pérdida (que en este caso se trata de las probabilidades obtenidas con el predictor) para resolver el árbol de juego, la principal complicación para imitar este sistema recomendación en otros juegos sería la obtención de esa función de pérdida. Este es un proceso costoso ya que normalmente los desarrolladores buscan hacer su juego lo más equilibrado posible en el que todos los equipos tengan la misma probabilidad de ganar.

7.2. Líneas Futuras

A continuación se listan algunas líneas de investigación que pueden desarrollarse en el futuro para continuar con el trabajo hecho en este proyecto:

- La precisión obtenida con el clasificador, que por criterios de diseño usa como características exclusivamente las estadísticas de los campeones, ha sido relativamente baja (alrededor del 53 %). Como se comentó en el estado del arte, en artículos como [19] [20] se incluyen en el predictor otra serie de características (por ejemplo el winrate de cada campeón o indicadores sobre los distintos jugadores de la partida, entre otros) con las que se reportan precisiones entorno al 60-70 %. Se propone como línea futura estudiar las ventajas e inconvenientes de utilizar esas u otras características en el clasificador.

- Extracción de nuevos datasets con datos de partidas de otros parches del juego para valorar si los resultados del predictor se siguen manteniendo entre cambios de versiones el juego.

- Los distintos parámetros seleccionados del Perceptrón Multicapa no se validaron con validación cruzada porque excedía el alcance del proyecto, y aunque todo parece indicar que los resultados con Keras no van obtener ventajas significativas con respecto a Scikit-Learn, para obtener esa conclusión con más certeza quedaría como línea futura hacer una exploración más sistemática en la que se valide cada una de las variables de la red neuronal.

- Buscar más estrategias para reducir el tiempo de computación del sistema recomendación con una cantidad suficiente de muestreo, de manera que se pueda aumentar la velocidad del algoritmo pero sin perder fiabilidad en la recomendación.

- Explorar más técnicas de teoría de juegos para la recomendación de campeones, como puede ser el Árbol de Búsqueda Monte Carlo.

- Dentro del propio algoritmo de recomendación, valdría la pena explorar el compromiso entre tener más profundidad en el árbol de juego, pero tener menos extensión en cada nivel del árbol (número de ramas).

- Usar la misma filosofía seguida en este proyecto para crear sistemas de ayuda en otros juegos que tengan fases previas a la partida en las que ocurra la selección de personajes (como pueden ser Dota 2, Heroes of the Storm o Battlerite, entre otros).

- Como se detalló, la validación del sistema de recomendación es una línea futura importante. Una vez validado, otra línea futura puede ser la implementación del sistema de recomendación en una aplicación que cualquier usuario pueda utilizar, de manera que no tenga la necesidad de instalar Python para usar el recomendador.

8. MARCO REGULADOR Y ENTORNO SOCIO-ECONÓMICO

8.1. Entorno socio-económico

El impacto socio-económico fue descrito con detalle en el Marco y Motivación del proyecto (1.1) , de donde se obtiene como conclusión que la industria de los videojuegos, y particularmente los eSports (en los cuales League of Legends domina el panorama mundial), está en continuo crecimiento y con una gran cantidad de profesionales, una audiencia similar o superior a deportes populares y una cantidad de jugadores activos increíblemente grande, a la que va dirigida el sistema de recomendación diseñado en este proyecto.

8.1.1. Presupuesto

A continuación se presentan el presupuesto del proyecto, que se puede dividir en costes materiales, costes de personal y coste total.

Costes materiales

Para calcular el coste real de los objetos que pueden ser reutilizados, se utiliza el cálculo de amortización de la ecuación 8.1.

$$Coste\,imputable = \frac{A}{B} \cdot C \cdot D \tag{8.1}$$

donde A es el número de meses de uso, B es la vida útil, C es el coste del equipo (sin IVA) y D es el porcentaje de uso que se le dedica al proyecto. Los costes materiales asociados en el proyecto vienen especificados en la tabla 8.1

TABLA 8.1: COSTES MATERIALES

Descripción	Precio	Tiempo de uso	Vida útil	% de uso	Coste imputable
Ordenador personal	800€	10 meses	60 meses	70 %	93,33€
Software	0€	10 meses	-	100 %	0€
Material de oficina	25€	-	-	-	25 €
Tarifa de Internet	50€/mes	10 meses	-	25 %	125€
Tarifa de Luz	70€/mes	10 meses	-	20 %	140€
				TOTAL	383.33€

El software utilizado es Spyder (para la programación en Python) y Texmaker (para escribir esta memoria en LaTeX). Las librerías (y su versión) usadas en Python son: scikit-Learn (0.19.0), Keras (2.1.4) con tensorflow (1.2.1) por debajo, matplotlib (1.5.1), numpy (1.13.1), pandas (0.18.1), requests (2.14.2), además de otra serie de librerías estándar de Python (como random, pickle o bisect, entre otras).

Costes de personal

En este proyecto han trabajado dos personas:

- Sergio Elola García: la dedicación al proyecto ha sido de 3 horas diarias, 5 días a la semana, durante un total de 10 meses. Con un total de 640 horas y unos honorarios de 15€/hora, da lugar a un coste total de 9600€.
- Jesús Cid-Sueiro: la cantidad de tiempo invertido entre tutorías y correcciones asciende a 80 horas, con unos honorarios de 35 €/hora da lugar a un gasto total de 2800€.

En la tabla 8.2 se resumen los gastos asociados al personal.

TABLA 8.2: COSTES PERSONALES

Persona	Honorarios	Nº horas	Total
Sergio Elola García	12€	630	7560€
Jesús Cid-Sueiro	30 €	80	2400€
		TOTAL	9960€

Presupuesto total

El presupuesto total se muestra en la tabla 8.3. Se trata de la suma de los costes materiales y costes personales, a la que se le aplica el IVA (21 %).

TABLA 8.3: COSTES TOTAL

Descripción	Coste imputable
Costes materiales	383,33€
Costes de personal	9960€
IVA (21 %)	2172,1€
TOTAL	**12515,43€**

El presupuesto final del proyecto asciende a DOCE MIL QUINIENTOS QUINCE EUROS.

8.2. Marco Regulador

Los datos usados en este proyecto han sido extraídos exclusivamente de la API de Riot Games. Esta API tiene sus propias políticas generales [46] en las que se insta usar la API para proporcionar una mejor experiencia de juego. Para poder usar la API es necesario aceptar sus Términos de Uso [47]. Aceptando estos términos, se concede un acceso limitado, no exclusivo, no sublicenciable, no transferible y revocable a los materiales propios de Riot Games.

Entre las condiciones a las que el usuario está sujeto se incluyen temas relacionados con el comportamiento apropiado del usuario, respeto de los derechos intelectuales (tanto de Riot games como de terceros), respeto de la privacidad o compromiso con la seguridad de las aplicaciones que usen la API. Además, está totalmente prohibido obtener dinero por proporcionar servicios de acceso exclusivo a características basadas total o parcialmente en datos obtenidos de la API.

El **Reglamento General de Protección de Datos (RPGD)**, que entró en vigor en mayo de 2016 y es aplicable desde mayo de 2018, es el que regula el tratamiento de datos personales. Cualquier desarrollador que use la API puede tener acceso a datos personales sujetos al RPGD [47]. Los datos usados para este proyecto (sección 3.2) respetan íntegramente el RPGD y el código compartido no incluye datos extraídos con la API.

BIBLIOGRAFÍA

[1] T. Wijman, *Mobile Revenues Account for More Than 50 Percent of the Global Games Market as It Reaches 137.9 Billion Dollars in 2018*, 2018. [En línea]. Disponible en: `https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/`, último acceso el 22/09/2018.

[2] J. Hamari y M. Sjöblom, "What is eSports and why do people watch it?", en *Internet Research, Vol. 27 Issue: 2, pp. 211-232*, 2017.

[3] J. Pannekeet, *Global Esports Economy Will Reach 905.6 Million Dollars in 2018 as Brand Investment Grows by 48 Perfect*, 2018. [En línea]. Disponible en: `https://newzoo.com/insights/articles/newzoo-global-esports-economy-will-reach-905-6-million-2018-brand-investment-grows-48/`, último acceso el 22/09/2018.

[4] P. Tassi, *Riot Games Reveals 'League of Legends' Has 100 Million Monthly Players*, 2016. [En línea]. Disponible en: `https://www.forbes.com/sites/insertcoin/2016/09/13/riot-games-reveals-league-of-legends-has-100-million-monthly-players/#672b5e495aa8`, último acceso el 22/09/2018.

[5] Raizin y Sameboat, *Map of MOBA.png*, 2013. [En línea]. Disponible en: `https://commons.wikimedia.org/w/index.php?curid=29443207`, último acceso el 22/09/2018.

[6] S. Ferrari, "From Generative to Conventional Play: MOBA and League of Legends", *DiGRA International Conference*, 2013.

[7] R. Lara-Cabrera, C. Cotta y A. J. Fernández-Leiva, "A review of computational intelligence in RTS games", *IEEE Symposium on Foundations of Computational Intelligence*, 2013.

[8] M. Buro, "Real-Time Strategy Games: A New AI Research Challenge", *18th International Joint Conferences on Artificial Intelligence*, pp. 1534-1535, 2003.

[9] H. Chan, A. Fern, S. Ray, N. Wilson y C. Ventura, "Online Planning for Resource Production in Real-Time Strategy Games", *Association for the Advancement of Artificial Intelligence*, 2007.

[10] F. Schadd, E. Bakkes y P. Spronck, "Opponent modeling in real-time strategy games", *Proceedings of the GAME-ON 2007*, pp. 61-68, 2007.

[11] S. Ganzfried y T. Sandholm, "Game Theory-Based Opponent Modeling in Large Imperfect-Information Games", *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, pp. 2-6, 2007.

[12] P. Spronck, I. Sprinkhuizen-kuyper y E. Postma, "Online adaptation of game opponent ai in simulation and in practice", *In Proceedings of the Fourth International Conference on Intelligent Games and Simulation*, pp. 93-100, 2003.

[13] F. Kabanza, P. Bellefeuille, F. Bisson, A. R. Benaskeur y H. Irandoust, "Opponent Behaviour Recognition for Real-time Strategy Games", *Proceedings of the 5th AAAI Conference on Plan, Activity, and Intent Recognition*, pp. 29-36, 2010.

[14] K. D. Forbus, J. V. Mahoney y K. Dill, "How qualitative spatial reasoning can improve strategy game AIs", *IEEE Intelligent Systems*, vol. 17, n.º 4, pp. 25-30, 2002.

[15] T. Kaukoranta, J. Smed y H. Hakonen, "Role of Pattern Recognition in Computer Games", *Proceedings of the 2nd International Conference on Application and Development of Computer Games*, pp. 189-194, 2003.

[16] D. W. Aha, M. Molineaux y M. ponsen, "Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game", *Proceedings of the Sixth International Conference on Case-Based Reasoning*, pp. 5-20, 2005.

[17] G. Synnaeve y P. Bessière, "A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft", *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 79-84, 2011.

[18] D. C. Cheng y R. Thawonmas, "Case-based Plan Recognition for Real-Time Strategy Games", *Proceedings of the 5th International Conference on Case-based Reasoning: Research and Development*, pp. 161-170, 2003.

[19] H. Y. Ong, S. Deolalikar y M. Peng, "Player Behavior and Optimal Team Composition in Online Multiplayer Games", 2014.

[20] J. Yin, "Predicting League of Legends Ranked Match Outcomes With Machine Learning", 2018.

[21] M. S. Cabrera, "Predicting the winner of a League of Legends match at the 10-Minute Mark", 2016.

[22] B. Fradet, *Using Spark and Kafka for the 2016 Riot Games hackathon*, 2016. [En línea]. Disponible en: `https://benfradet.github.io/blog/2016/10/22/riot-games-2016-hackathon`, último acceso el 22/09/2018.

[23] Y. N. Ravari, S. Bakkes y P. Spronck, "StarCraft Winner Prediction", *AIIDE-16, Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2016.

[24] H.-S. Park, H.-C. Cho, K. Lee y K.-J. Kim, "Prediction of Early Stage Opponents Strategy for StarCraft AI Using Scouting and Machine Learning", *Proceedings of the Workshop at SIGGRAPH Asia*, pp. 7-12, 2012.

[25] V. da Costa Oliveira, B. J. Placides, M. de Freitas Oliveira Baffa y A. F. da Veiga Machado, "A Hybrid Approach To Build Automatic Team Composition In League of Legends", *Proceedings of SBGames*, 2017.

[26] Z. Chen et al., "The Art of Drafting: A Team-Oriented Hero Recommendation System for Multiplayer Online Battle Arena Games", 2018.

[27] R. Bhattacharya y A. Sabik, "Data-driven Recommendation Systems for Multiplayer Online Battle Arenas", 2015.

[28] D. P. Kevin Conley, "How Does He Saw Me? A Recommendation Engine for Picking Heroes in Dota 2", 2014.

[29] A. Summerville, M. Cook y B. Steenhuisen, "Draft-Analysis of the Ancients: Predicting Draft Picks in DotA 2 Using Machine Learning", *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2016.

[30] J. Cid-Sueiro y J. A. García, *ML4all (Machine Learning for all) - Logistic Regression*, 2016. [En línea]. Disponible en: `https://github.com/ML4DS/ML4all/tree/master/C3.Classification_LogReg`, último acceso el 22/09/2018.

[31] J. Brownlee, *Feature Selection For Machine Learning in Python*, 2016. [En línea]. Disponible en: `https://machinelearningmastery.com/feature-selection-machine-learning-python/`, último acceso el 22/09/2018.

[32] Scikit Learn developers, *Support Vector Machines*, 2007-2017. [En línea]. Disponible en: `http://scikit-learn.org/stable/modules/svm.html`, último acceso el 22/09/2018.

[33] ——, *Naive Bayes*, 2007-2017. [En línea]. Disponible en: `http://scikit-learn.org/stable/modules/naive_bayes.html`, último acceso el 22/09/2018.

[34] ——, *Decision Trees*, 2007-2017. [En línea]. Disponible en: `http://scikit-learn.org/stable/modules/tree.html`, último acceso el 22/09/2018.

[35] ——, *Ensemble methods*, 2007-2017. [En línea]. Disponible en: `http://scikit-learn.org/stable/modules/ensemble.html`, último acceso el 22/09/2018.

[36] K. P. Murphy, *Machine Learning: A Probabilistic Perspective, chapter 28*. The MIT Press, 2012.

[37] A. Cartas, *File:Perceptrón 5 unidades.svg*, 2015. [En línea]. Disponible en: `https://commons.wikimedia.org/wiki/File:Perceptr%C3%B3n_5_unidades.svg`, último acceso el 22/09/2018.

[38] P. I. Viñuela e I. M. Galvan, *Redes de neuronas artificiales: un enfoque práctico*. Pearson Education, 2004.

[39] H. Paz, Y. Jimenez y A. Larco, "Desarrollo de un sistema inteligente para la clasificación de documentos ya digitalizados aplicando redes neuronales supervisadas", *Revista Tecnológica ESPOL – RTE, Vol. 28, N. 1, 8-23*, 2015.

[40] I. Goodfellow, Y. Bengio y A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org, último acceso el 22/09/2018.

[41] G. Cybenko, "Approximation by Superpositions of a Sigmoidal Function", *Neural Networks, 4(2), 251–257*, 1989.

[42] S. Ruder, *An overview of gradient descent optimization algorithms*, 2016. [En línea]. Disponible en: http://ruder.io/optimizing-gradient-descent/, último acceso el 22/09/2018.

[43] S. Russell y P. Norvig, *Inteligenica Artificial. Un enfoque Moderno*. Pearson Education, 2ª ed. 184-186, 2004, Madrid.

[44] League of Legends wiki, *List of Champions Positions*. [En línea]. Disponible en: http://leagueoflegends.wikia.com/wiki/List_of_champions/Position, último acceso el 22/09/2018.

[45] D. E. Knuth y R. W. Moore, *An Analysis of Alpha-Beta Pruning*. Artificial Intelligence Vol. 6, No. 4: 293-326, 1975.

[46] Riot Games API, *General Policies Riot API*. [En línea]. Disponible en: https://developer.riotgames.com/policies.html, último acceso el 22/09/2018.

[47] ——, *API Terms and Conditions*. [En línea]. Disponible en: https://developer.riotgames.com/terms, último acceso el 22/09/2018.

ANEXO A: ENGLISH SUMMARY

Introduction and objectives

The video game industry is growing at a high level. According to Newzoo's estimates [1], the global revenues will grow to $180 billion by 2021, that is, a growth of 11 %. Together with the Mobile Gaming Market, one of the big reasons for this growth is the emergence of eSports. We refer to eSports as competitive video games in which teams are coordinated by different leagues and tournaments, and where professional players belong to teams or other organizations that are sponsored. The Global eSports Industry is expected to reach $905.6 million in 2018, which represents a year-on-year growth of 38 % [3]. However, eSports are characterized for the large amount of viewers they have, around 335 million viewers in 2017. One of the most popular eSports games, if not the most, is League of Legends (LoL). Even if there is no official information, it is believed with some confidence the amount of monthly players exceeded 100 millions by 2016 [4]. That is main reason why LoL can be so interesting to investigate on.

LoL is a game in which two teams (blue and red), made up of 5 players each, fight against each other and the main goal is to destroy the main opponent structure. Each player controls a different character (called "champion"), and every champion has its own different abilities and base statistics (which are the values that indicate how good is a champion in something, like for instance how much life a champion has or how fast it can run). In every match of League of Legends (in PvP mode), it exists a pre-game phase called "Draft" or "Champion Selection" in which every player select its own character ("champion") to play. The Draft works as follows: first, there is the "ban phase", where every player bans a champion so no one in the game (including its own team) can play that champion. After the ban phase, the "selection phase" takes place, and it is made in a very specific way. There are 6 waves: in the first wave, the first player of the blue team selects the champion. In the second wave, the first and second players of the red team select at the same time. In the third one, the second and third players of the blue team select at the same time, and so on. This phase is represented in figure A.1.

The decisions made by the players in the champion selection are key to winning the game. Those decisions are a hard problem and there are no automatic tools that facilitate this decision making, which is why in most cases the decisions are made by intuition acquired through experience in the game. This type of selections phases are very difficult to formalize, that is to say, there is no method or formula that allows to obtain the best decision in each case. Nevertheless, there is a large amount of data that could potentially allow inductive learning from the results of previous games. The motivation of this project is to explore that possibility, so a system can be designed in order to help ordinary LoL players make better decisions in their own

games.

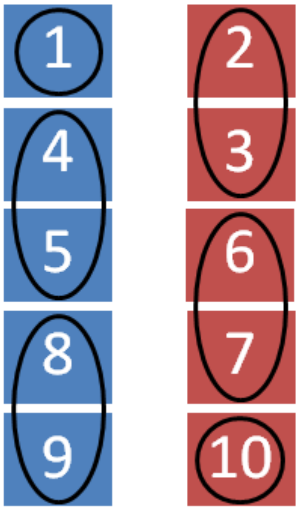


Figure A.1: Draft representation (champion selection). Circle and elipses represent the selection order (each wave). Numbers identify each different player.

The objective of this project is therefore to design a recommendation system for any player in the Draft phase of a League of Legends game, based on the champions that have been previously selected. The idea is to first elaborate a model that can predict the winner of a game given the 10 champions that make a complete game, and then use it for making recommendations using game theory techniques (Minimax algorithm). The predictor will be a Machine learning model trained with data from previous games. Two main separate strategies will be followed for the designing of the model: a Classical Machine Learning strategy (that is, all approaches of Machine Learning that do not include Deep Learning), and a Deep Learning strategy.

State of the art

League of Legends belongs to a video game genre called MOBA (Multiplayer Online Battle Arena), which is at the same time a sub-genre of RTS games (Real Time Strategy). RTS games, as its own name suggests, are characterized by being strategy games that are developed over time, in other words, they do not progress incrementally in turns. There is a lot of Artificial Intelligence applications for RTS Games, which make this a recent field of research. There are many challenges in RTS, which are explained in Michael Buro [8] and Raúl Lara-Cabrera et al. [7] papers. Among all the challenges, even if the Draft cannot be considered technically as a real-time phase (since it is done in turns), we could label this project as an Opponent

Modeling challenge because the decisions made during the champion selection are essential for the game. We could also consider it as a Collaboration challenge as the main goal is to build the best possible team.

The predictor is the most important part of the recommendation system. The two most important works related with this task are the Hao Yi Ong et al [19] paper and the Jihan Yi [20] paper. The first obtains 70 % accuracy with both SVM and GDA (but being SVM much slower than GDA). The second one gets 60 % accuracy using Random Forest and Gradient Boosting. However, to train the models, both articles use as features some information about players (for instance the win ratio, number of times the player has won in the 15 recent games, average damage...) that is not possible to obtain in real-time during the Draft (at least not for the enemy players, as it is not possible to see the enemy names until the game starts). The idea for this project is creating a predictor based only on the statistics of the champions (life, attack damage, armor...). This way, although the accuracy of the predictor is expected to be much lower, we can assure that it is going to be possible to make predictions in real time and therefore we will be able to use those predictions for making recommendations.

With regard to the recommendation system, there are several related works made for MOBA games (especially for Dota 2), as these types of games usually have similar character selection phases. But in particular for LoL, in [25] the authours propose a very similar system as the one that is going to be developed in this project. They suggest to "create a game tree to model all champions possibilities, evaluate every team with a linear regression function for giving a victory rate, and use Minimax algorithm with Alpha-Beta Pruning to optimize the champion recommendation and minimize the loss". The goal of this project is to study the different models for the predictor and select the best one, and also design a final optimal algorithm that can be used to provide a recommendation in real time. The development of a final algorithm, as will be seen further on, requires more in-depth study than the one that is done in that article.

Variety, extraction and distribution of data

The Riot Games API, a free JSON-based API (Application Programming Interface), is the tool used to extract the data that will be used to build the predictor. It requires an API Key to be used. As explained, the data used to train the model are the statistics of each champion selected in each game. The statistics of the champions remain fixed for the same patch (version of the game), so downloading them once per patch is enough. There are 21 different possible statistics for each champion, of which we will use 18 of them (as the 2 other have zero value for almost all the champions).

The information that will be saved for each game is made up of just 11 numbers. The first number indicates the winning team of the game (0 for blue, 1 for red), the next 5 numbers are the champions of the blue team and the 5 final numbers are the champions of the red team. Besides, the champions of each team are ordered always according to their role in the game: Top, Jungle, Middle, ADC and Support. In LoL, each role has a different position on the map and its function in the game, so it is important to save the games in this order so the predictions are more accurate.

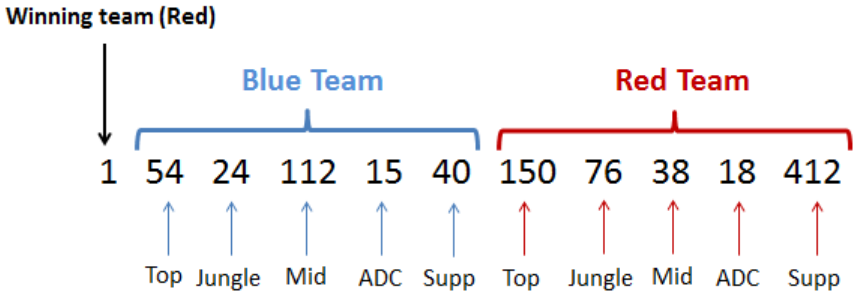


Figure A.2: Diagram of the data saved per game

Hence, as there are 10 champions per game and 18 statistics per champion, the data that will be used to train the model will have an input dimension of 180 (that is, the number of features will be 180). If all the statistis are not used, then the input dimension will be $10 \cdot n$, where n is the number of selected features.

Instead of using external libraries for the extraction of the data with the API (such as "Cassiopeia" in Python), a simple algorithm is created to be able to extract data continuously. It is based on the fact that it is possible to request a list of games in a certain period of time about a player account, and when information about a game is requested, new player accounts (of the other participants in the game) are discovered. Thus, a loop is created for a non-stop extraction of the games. In addition, each game is classified in a league according to the leagues of the different players of the game (as players who play in "ranked mode" are classified depending on how good they are). The reason for classifying the games is that the level of players changes a lot between different leagues.

A total of 337100 games were obtained from the LoL patch 8.3.1 (from 7 to 22 February 2018), together with the statistics of each champion (there are 139 champions available in this patch). Since the idea is to use the data for training and evaluate the different models, it is common in Machine Learning to split the dataset into different sets. It is decided to divide the data into training set (used for training the model, that is to say, to adjust the parameters of the different models), validation set (to estimate how each model has been trained) and test set (used exclusively to test the final model). The distribution of the dataset is the following: 50 % of the total games for the training set, 30 % for the validation set and 20 % for the test set.

Prediction with Classical Machine Learning

The prediction of the winning team of a game is a problem of binary classification because an attempt will be made to identify to which category each game belongs, being two possible categories: the winner is the blue team or the winner is the red team. *Scikit-Learn*[1] is the Python tool that will be used for creating all the models in Classical Machine Learning. In order to be able to add improvements and test different parameters in a model, it is necessary to start from an initial model. This basic model has the following characteristics: an algorithm of Logistic Regression with cross validation for C parameter is used, all the possible features (180) are used, and no process is apply to the data (like for example data normalization). The accuracy obtained with this model is around 52-53 % for all leagues.

Several data pre-processing techniques are now tested. Pre-processing techniques are very common in Machine Learning and they have many benefits such as reduction of overfitting, improvement of accuracy and reduction of training time. First of all, multiple normalization algorithms of Scikit-Learn are tested, being *MaxAbsScaler* the chosen one (it increases previous accuracy by up to 1 %, that although it may seen little, it is enough for such a difficult problem of classification). It is important to note that normalization is a process that must go through just before training the model, i.e. at the end of all the data pre-processing and also that the same transformation must be applied for the training and the validation data (usually transformation parameters are fit using the training set and then they are used to transform also the validation and test set).

The second data pre-processsing technique is Feature Extraction. A data transformation is proposed for being able to reduce the number of features and to make the model understand that in each game there are two teams and that there are 5 players in each team. Unfortunately, this transformation is not successful and it is decided not to include it in the model. The last technique used is Feature Selection and four different strategies are tested (Univariate Feature Selection, PCA, Recursive Feature Elimination and Feature Importance). Univariate Feature Selection (*SelectKBest* with *f_classif* score function in Scikit-Learn) is selected as the best for being fast and providing good precision results. However, the precision decreases as less features are used. That is why it was decided that all the possible features will be used but, in case it is necessary (for instance to increase the processing speed), it is possible to reduce the number of features without losing much accuracy.

Now, the goal is to study different classification algorithms and to evaluate how they respond to the problem. There are many algorithms, and the aim is to analyze at least one of each class. The studied algorithms are: Support Vector Machines

[1] Available at `http://scikit-learn.org/stable/`

(SVM), k Nearest Neighbors (kNN), Naive Bayes, Decision Trees, Random Forest and Gradient Boosting. SVM and kNN they both have quadratic time complexity, and since there is a lot of training data and the input dimension is so large, the models take too much to train. That is why *LinearSVC* is used for SVM (as it scales better with the number of samples). However, kNN does not have a similar algorithm that with less time complexity, so *SelectKBest* is used for feature selection but it is still very slow when predicting new samples, so the use of kNN is discarded. Using the default model of Decision Trees, the model got 100 % train accuracy, but close to 50 % validation accuracy, so overfitting was occurring. In order to reduce the overfitting, some parameters of the model were modified, and the parameter that worked best was the maximum depth of the tree. Random Forest and Gradient Boosting are two ensemble algorithms that use many estimators (Decision Trees), so the parameters for number of estimators and maximum depth had to be adjusted (for Gradient Boosting the default value, max_depth = 3, is used because it provides the best accuracy). The table A.1 shows the best results for each algorithm, taking into account that all the possible parameters have been studied and adjusted.

TABLE A.1: VALIDATION ACCURACY BY CLASSIFICATION ALGORITHM

Liga	Log. Regr.	SVM	N. Bayes	Dec. Tree	R. Forest	Grad. Boost
Bronce	0.5404	0.5434	0.5237	0.5205	0.5337	**0.5472**
Plata	0.5351	0.5345	0.5253	0.5168	0.5349	**0.5450**
Oro	0.5337	0.5342	0.5170	0.5184	0.5301	**0.5404**
Platino	0.5326	0.5316	0.5222	0.5158	0.5303	**0.5331**
Diamante	0.5281	0.5277	0.5260	0.5214	**0.5326**	0.5323
Challenger	0.5234	0.5371	0.5399	0.5262	**0.5510**	0.5289

Finally, the selected model for the Classical Machine Learning approach of the predictor is Gradient Boosting, with exponential loss function and different number of estimators for each league (that can be found in table 4.10).

Prediction with Deep Learning

Deep Learning is a branch of Machine Learning that with the use of specific structures called “Neuronal Networks” it is able of increasing the level of abstraction of the data. The neural network structure that best fits the problem and that will be used is the “Multilayer Percetron”. A Multilayer Perceptron is the combination of several perceptrons, where a perceptron is an algorithm that works very similar the logistic regression: it fits a vector of weights and the result of the scalar product between the input vector and the weights vector is introduced to an activation function that returns the result of the prediction. The perceptrons are grouped into

several layers: input layer (in which the neurons receive the signals and propagate them to the rest of the network, so they do do not properly work as neurons), hidden layers (in which the neurons process the received signals and transmit the result to the next neurons) and the output layer (that returns the result of the whole network). For the design of the Multilayer Perceptron, the Python library *Keras*[1] will be used.

Again, a basic model of a Multilayer Perceptron will be created from which improvements will be added. Two main ideas arise for the main structure: a very simple one that is formed by a single neuron (which is equivalent to an algorithm of what was called Classic Machine Learning), and a more complex one that tries to take advantage of the player roles and of the fact that all champions have the same statistics (with different values each one). Interestingly, the simple model provides better accuracy than the complex one, obtaining very similar results to those of Logistic Regression (as expected, as it consists of a single neuron). The first model is therefore chosen as the basic model.

Improvements are now made the basic model. First of all, the batch size (amount of samples used in each iteration) is set to 256 (for being an intermediate value and a power of two). Next, the number of hidden layers and number of neurons per layer are studied. After an extensive testing process using a lot of possible combinations, it is concluded that that a single hidden layer is enough (more layers causes overfitting and a longer training time), and the obtained accuracy is very similar to having no hidden layer. The best number of neurons in that single hidden layer is between 50 and 200. The activation function that better worked for all the neurons was the *sigmoid* function. Additionally, another possible improvement is to increase the symmetry of the model. This means that if in a game the champions of each team are switched (that is to say, the champions of the red team are changed by the ones of the blue team, and vice versa), the result of new game must be the opposite. To do this, new games are creating performing that operation (switch the teams and change the result) and they are added to the rest of the dataset. The results are very similar to the previous obtained results (and slightly worse), which may indicate that the model is intrinsically learning that symmetrical capacity.

The last improvement being studied is the optimization algorithm of the neural network, which is the one responsible of minimizing the error function in order to get the optimal values for the weights in the neurons. Until now, the *RMSProp* algorithm had been used, but Keras provides more algorithms: *Adagrad*, *Adelta*, *Adam*, *Adamax* and *Nadam*. The accuracy for each algorithm, using a Multilayer Perceptron with a single hidden layer with 50 neurons and one output layer with one neuron (all neurons use activation function *sigmoid*), can be found in table A.2.

[1] Available at `https://keras.io/`

TABLE A.2:VALIDATION ACCURACY BY OPTIMIZATION ALGORITHM

Liga	RMSprop	Adagrad	Adadelta	Adam	Adamax	Nadam
Bronce	0.5457	0.5428	0.5420	0.5451	0.5437	**0.5458**
Silver	0.5380	0.5365	0.5365	0.5386	0.5368	**0.5394**
Gold	0.5342	0.5326	0.5338	0.5352	0.5363	**0.5374**
Platinum	0.5335	0.5321	0.5346	**0.5348**	0.5338	0.5332
Diamond	0.5302	0.5272	0.5278	0.5306	**0.5308**	0.5299
Challenger	0.5321	0.5381	0.5234	0.5271	0.5201	**0.5383**

All the algorithms have similar results, but we obtain the following conclusions: *Adagrad* and *Adadelta* perform similarly as the second is an extension of the first, and they both require a little more time of training than the rest of algorithms. *Adam* and *Adamax*, since one is a variant of the other, they also obtain similar results and they both converge very quickly. *Nadam* is also a very similar algorithm to these two previous ones, but it takes longer to converge and it does not get similar results to them.

The final model selected for Deep Learning approach of the predictor is a Multilayer Perceptron composed by one hidden layer with 50 neurons, the output layer with one neuron (all neurons with activation function *sigmoid*), and *Adagrad* as optimization algorithm.

Recommendation Algorithm

Before starting to explain the final algorithm, it is important to explain how a game tree and the Minimax algorithm work. Every turn-based game can be represented as a game tree (very similar to a decision tree), where the nodes represent a point where a decision has to be made, and each of the branches represent a possible decisions. A terminal node (leaf), represents a class or result. The Minimax is an algorithm that minimizes the possible loss for a worst case scenario. So the minimax method can be summarized as "how to choose the best decision for you assuming that your opponent will choose the worst for you". Having a loss function that can get a possible value for any layer of the tree, a decision can be found using the Minimax algorithm.

The Draft phase, as it is done in turns (figure A.1), can be represented as a game tree. The loss function required to use the Minimax algorithm is the predictor that has been designed in previous chapters. As there are 139 possible champions, the game tree would be huge for most of the players in the Draft. As it is supposed to work in real-time, sampling must be used in order to be able to provide a recommendation in the available time of each wave (30 sec). A sampling function called

get_probs() is created. This function receives a list of the selected champs, a list of possible champs to be chosen, the number of samples to be made for each champion and (optionally) the role that player is going to play (top, jungle, etc.). The function uses sampling for returning a vector with all the probabilities of each champion (of the list of possible champs) to win the game. This function will be used later in the final algorithm.

The recommendation for the 10th player (figure A.1) is simple: each possible is introduced to the predictor along to the rest of the 9 selected champions, and the champion that gets the maximum probability to win the game is the one that will be recommended. For the 9th player, the recommendation is not that simple. A game tree similar to the one of the figure A.3 must be built, and apply the Minmax algorithm: for each champion that player 9 can choose, the process made for the champion 10th is done, but instead of selecting the maximum probability, as the enemy player will try to select the worst champion for you, a minimization must be done. Then, once all the minimum probabilities have been obtained for each possible champion, the maximum one among them is selected (that is to say, we select the best champion assuming that the enemy selects the worst champion for us). The champion corresponding to that probability will be the recommended champion. Assuming that 10 champions were banned at the beginning of the Draft, a number of $121 \cdot 120 = 14520$ predictions must be done, which is still affordable to do.

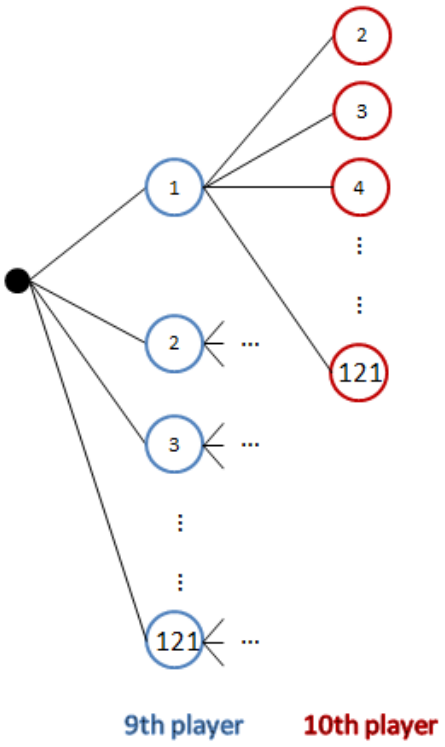


Figure A.3: Game tree for 9th player

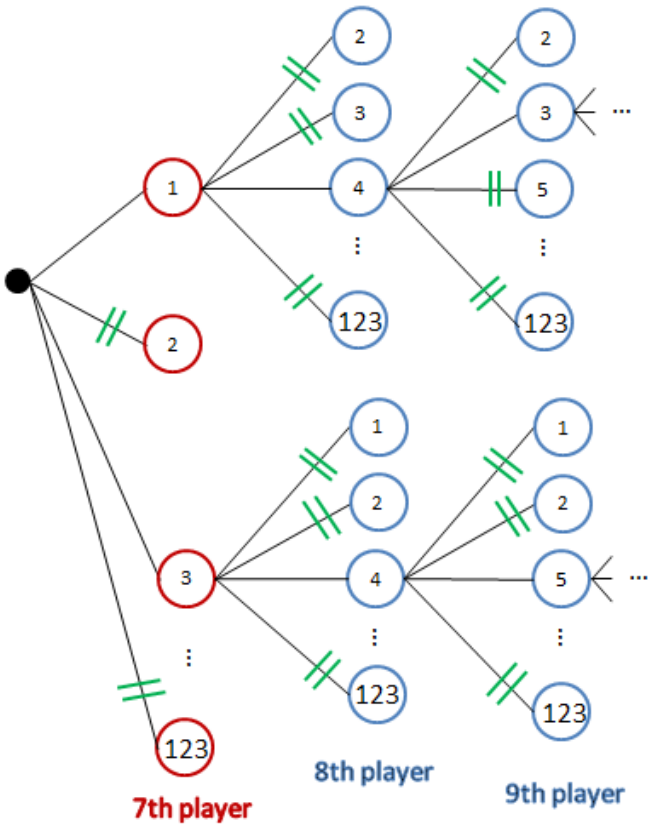


Figure A.4: Example of pruned tree

However, the recommendation of all the rest of the players (8t, 7th, 6th...) is not possible to do following the same strategy than with champion 9th, as for example the amount of predictions for the 8th champion is 1771440, or 217 million for the 7th champion, completely unaffordable to perform in real-time. That is why we use the previously created function *get_probs()* together with a technique called pruning (inspired by the Alpha-Beta pruning [45]). The idea is to eliminate (prune) those unnecessary branches in order to be able to represent more layers of the tree (the figure A.4 shows an example of a pruned tree). In each layer, the probabilities for each champion are obtained using the function *get_probs()*, and the tree is still being built but only on the champions that get bigger or smaller probabilities (depending on whether the layer of the tree has to be maximized or minimized, respectively). Finally, when a certain number of layers of the tree are built, the Minimax algorithm is used to get the optimal decision.

The final algorithm proposed constructs two layers (to reduce processing time as much as possible by having a sufficiently large number of samples). However, the elaboration of an algorithm with more layers is automatic. To request a recommendation the next parameters must be introduced:

- **list_selected**: list of selected champions and their roles (if known). If a role of an enemy champion is not known (normally they are identified by intuition,

as the champions have the same role), then is is automatically assigned based on the roles of the other selected champions.

- **list_bans**: list of banned champions
- **role**: the role for which we want to make the recommendation.
- **n_samples**: number of random games (samples) that will be generated for each champion with the function *get_probs()*.
- **n_branch**: number of branches that will have the first layer (or what is the same, the number of branches that will not be pruned)

The flowchart of the proposed final algorithm can be found in the figure A.5. It is based on everything that has been explained above.

It is important to emphasize that this algorithm is a prototype, and that it has not been validated. The main reason to not validate is the difficulty that it implies. One possible option is to assume that the probability of victory provided by the predictor is accurate (i.e., that the accuracies are perfect), and evaluate the system using simulations. Another option could be to survey real players to compare their decisions or opinions, which are based on their own experience within the game, with the recommendations provided by the algorithm.

In any case, validation of the recommendation system is a problem itself, and it is outside the scope of this project, so it is proposed as a future line of research.

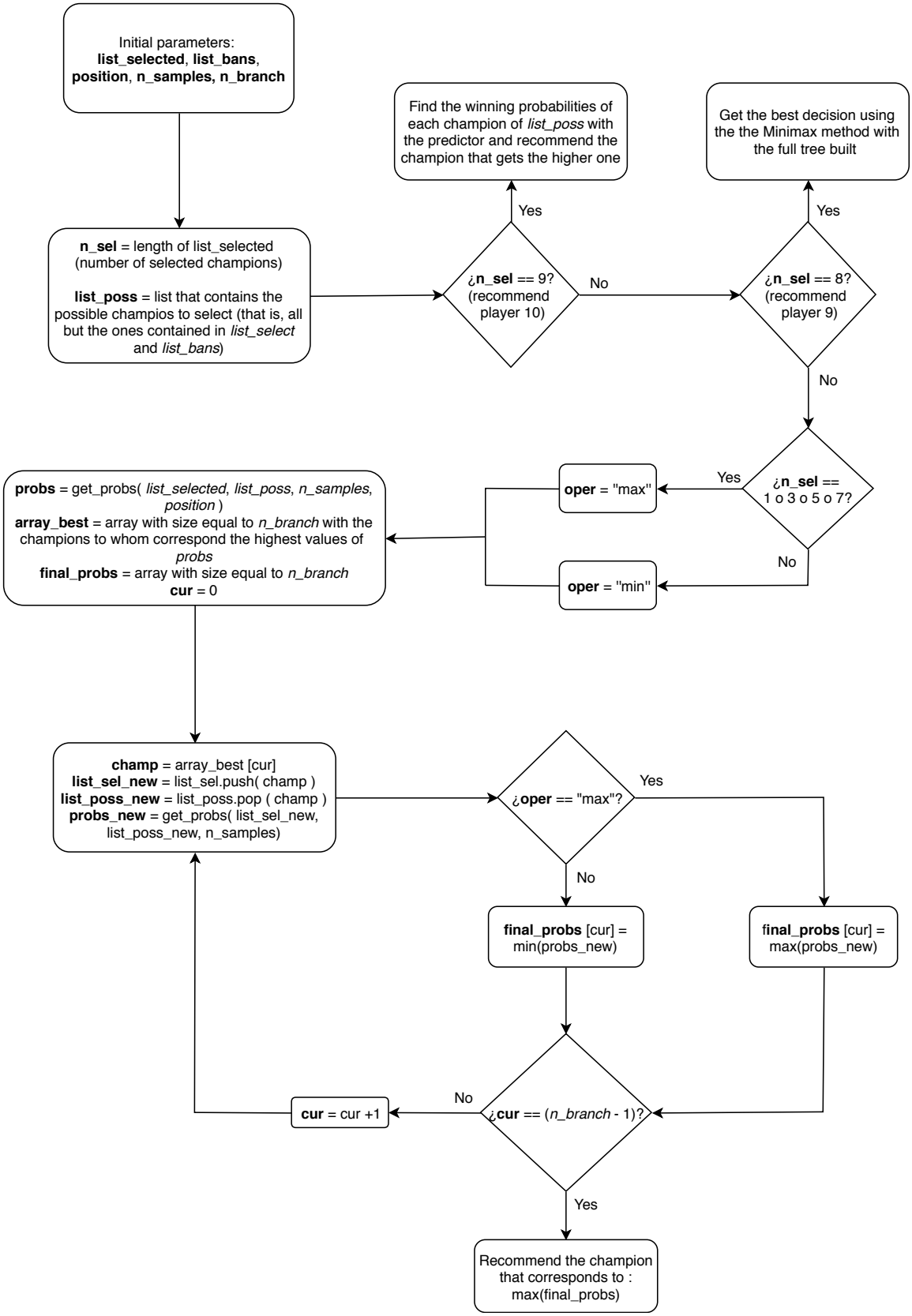


Figure A.5: Flowchart diagram of the recommendation algorithm for a tree with two layers

Conclusions and future lines

The objectives of the project have been achieved: a recommendation system for the champion selection phase of League of Legends has been designed. It is based on a predictor that is used for solving the game tree of the selection phase using the Minimax algorithm. In order to design this predictor, Machine Learning and Deep Learning strategies have been followed, providing the first best results. The difficulty of the problem makes the accuracy of the final predictor low. However, small differences with respect to randomness can provide advantages, and these are exploited by the recommendation system.

Among the contributions that have been made for this project, it stands out a method of data extraction through the API that classifies games as to the level of the players, a study of the behavior of different algorithms of Machine Learning for the prediction problem of the winner team and, above all, a proposal for a recommendation system based on game theory techniques that is in line with what is suggested in [25], and that can help players make decisions in real time.

It is also important to note that the system created can be applied to other video games with similar character selection phases. A good part of the software can be reused for other games, so it is decided to publish all the code of the project on `https://github.com/SergioEG/LoLRecommend` so that people can use it. The most difficult part for making a similar system in other games is the creation of a loss function (which in this case is the probabilities obtained with the predictor) that can solve the game tree. This is a difficult process because developers usually try to make their game as balanced as possible, in which all teams have the same probability of winning.

The future lines of research are listed below:

- The precision obtained with the classifier, which uses exclusively the champions statistics as features, has been relatively low (around 53 %). As it was discussed in the state of the art, in articles such as [19] [20], other features are included in the predictor and precisions around 60-70 % are reported. It is proposed as a future line to study the advantages and disadvantages of using those characteristics in the classifier.

- Extraction of new datasets with data from other games patches to assess whether the predictor results are still maintained between game version changes.

- The different parameters selected for the Multilayer Perceptron were not validated with cross validation because it exceeded the scope of the project, and although everything seems to indicate that the results with Keras are not going to obtain significant advantages with respect to Scikit-Learn, a future line of

research could be to carry out a more systematic exploration in which each of the variables of the neural network is validated.

- Search for more strategies to reduce the computation time of the recommendation system with a sufficient amount of sampling, so that the speed of the algorithm can increase without losing reliability in the recommendation.
- Explore more game theory techniques for recommending champions, such as the Monte Carlo Search Tree.
- Within the recommendation algorithm itself, it would be worth exploring the compromise between having more depth in the game tree, but having less extension at each level of the tree (number of branches).
- Use the same strategy followed in this project in order to design systems for other games that have similar pre-game phases in which the selection of characters occurs (such as Dota 2, Heroes of the Storm or Battlerite, among others).
- As detailed, the validation of the recommendation system is an important future line. Once validated, another future line can be the implementation of the recommendation system in an application that any user could use, so that they don't need to install Python to use the system.